BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

BACHELOR'S THESIS

Brno, 2021 Lukáš Mikan

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF MICROELECTRONICS
ÚSTAV MIKROELEKTRONIKY

AIR TRAFFIC SIMULATION WITH HACKRF ONE
SIMULACE LETOVÉHO PROVOZU S VYUŽITÍM HACKRF ONE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **Lukáš Mikan**
AUTOR PRÁCE

SUPERVISOR — **Ing. Martin Šťáva, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Microelectronics and Technology**

Department of Microelectronics

Student: Lukáš Mikan ***ID:*** 214957

Year of study: 3 ***Academic year:*** 2020/21

TITLE OF THESIS:

Air Traffic Simulation with HackRF One

INSTRUCTION:

Make yourself acquainted with the open-source software-defined radio (SDR) platform HackRF One. Also, get to know the basic operation of the MSS air surveillance system made by ERA, a.s. Look into the possibilities of using HackRF units to simulate air traffic, which can then be processed by the MSS. The input consists of aircraft track & speed data, and also contains precise coordinates of MSS's receiving antennas. At the output, there shall be radio signals containing ADS-B messages, as if generated by aircraft transponders. Each SDR unit shall transmit what a particular antenna would have received at its location, in the real world. Propose a solution and build a technical demo, suitable to be expanded upon and ultimately integrated into the development flow at ERA.

RECOMMENDED LITERATURE:

According to recommendations of supervisor

Date of project specification: 8.2.2021 ***Deadline for submission:*** 3.6.2021

Supervisor: Ing. Martin Šťáva, Ph.D.

doc. Ing. Jiří Háze, Ph.D.
Chair of study program board

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Mikroelektronika a technologie**

Ústav mikroelektroniky

Student: Lukáš Mikan ***ID:*** 214957
Ročník: 3 ***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Simulace letového provozu s využitím HackRF One

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s open-source software-defined radio (SDR) platformou HackRF One. Dále se seznamte s funkcí vzdušného přehledového systému MSS od firmy ERA, a.s. Prozkoumejte možnosti využití modulů HackRF pro simulaci letového provozu, který následně systém MSS dokáže zpracovat. Vstup je tvořen údaji o drahách a rychlostech letounů a také informací o přesné poloze přijímacích antén MSS. Výstupem bude rádiový signál nesoucí pravidelné zprávy standardu ADS-B používaného leteckými transpondéry. Signál z každého jednotlivého SDR modulu bude odpovídat tomu, který by konkrétní anténa reálně zachytila na své pozici. Navrhněte řešení a implementujte funkční technické demo vhodné pro další rozpracování a integraci do vývojového procesu v ERA.

DOPORUČENÁ LITERATURA:

According to recommendations of supervisor

Termín zadání: 8.2.2021 ***Termín odevzdání:*** 3.6.2021

Vedoucí práce: Ing. Martin Šťáva, Ph.D.

doc. Ing. Jiří Háze, Ph.D.
předseda rady studijního programu

ABSTRACT

This work deals with air traffic simulation in the context of a multilateration surveillance system. The technique of multilateration calculates aircraft position at the intersection of hyperboloids, obtained from the differences in time of arrival of transponder messages to multiple receivers at known coordinates. By substituting an SDR module for each receiving antenna, air traffic can be simulated with the use of made-up ADS-B messages which get sent to the various antenna ports with proper timing. This thesis uses the HackRF One SDR that had both its hardware and firmware modified in order to achieve tightly synchronized transmissions from potentially any number of interconnected HackRF units. The modifications described herein ensure an identical sampling clock, both frequency- and phase-wise, across all units, as well as simultaneous triggering of transmissions. In the second part of the thesis, an algorithmic solution is presented for assembling fictional air scenarios with arbitrary number of flights and receiving stations. The output is a set of data streams suitable to be transmitted through synchronized HackRF's. Each stream represents a specific vantage point, a specific antenna, and contains precisely timed ADS-B messages, encoded using pulse-position modulation and converted into IQ samples. The complete solution can be used to validate the functioning of a real multilateration system, such as the MSS made by ERA, a.s.

KEYWORDS

HackRF, SDR, multilateration, MSS, ERA, air traffic, ADS-B

ABSTRAKT

Práce se zabývá problematikou simulace letového provozu v kontextu multilateračního přehledového systému. Technika multilaterace počítá polohu letounu coby průsečík hyperboloidů na základě rozdílů v čase příchodu zpráv z palubního transpondéru na různé pozice přijímacích antén. Při nahrazení každé antény SDR modulem lze vzdušný provoz nasimulovat pomocí umělých ADS-B zpráv, které jsou ve správných časech přivedeny do jednotlivých anténních vstupů. V této práci je použito SDR HackRF One, jehož hardware i firmware byl modifikován pro dosažení úzce synchronizovaného vysílání z potenciálně libovolného počtu propojených HackRF jednotek. Zde popsané úpravy zajišťují shodný kmitočet i fázi vzorkovacího hodinového signálu na všech HackRF, stejně jako současné spuštění přenosu. Ve druhé části práce je představeno algoritmické řešení umožňující sestavit fiktivní vzdušný scénář s libovolným počtem letů i přijímacích antén. Výstupem je sada datových streamů vhodná pro vyslání skrz synchronizovaná HackRF. Každý stream odpovídá specifické poloze antény v krajině a obsahuje přesně načasované zprávy standardu ADS-B zakódovány pulzně-poziční modulací a převedeny na IQ vzorky. Celý systém umožňuje testovat správnou funkci reálného multilateračního sledovače, jako je například produkt MSS od firmy ERA, a.s.

KLÍČOVÁ SLOVA

HackRF, SDR, multilaterace, MSS, ERA, letový provoz, ADS-B

MIKAN, Lukáš. *Air Traffic Simulation with HackRF One*. Brno, 2021, 65 p. Bachelor's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Microelectronics. Advised by Ing. Martin Šťáva, Ph.D.

Typeset by the `thesis` package, version 4.00; `http://latex.feec.vutbr.cz`

ROZŠÍŘENÝ ABSTRAKT

Firma ERA, a.s. vyrábí pokročilý systém pro sledování letového provozu MSS (Multisensor Surveillance System). Tento systém se principem funkce řadí mezi tzv. pasivní multilaterační sledovače. Narozdíl od konvenčního radaru nevysílá žádné signály do éteru, nýbrž pouze naslouchá příchozím zprávám od letounů. Instalace MSS se skládá alespoň se tří základnových stanic, které musí být umístěny na přesně známých pozicích. Na základě rozdílů v čase příchodu (TDOA, Time Difference of Arrival) stejných zpráv na jednotlivé stanice vypočítává centrální server MSS polohu letounů v reálném čase. Výstupem serveru jsou sledovací data ve standardním formátu ASTERIX (All Purpose Structured Eurocontrol Surveillance Information Exchange), jež zákazník (letiště, národní středisko ATC) obvykle napojí do své další instrumentace a jejichž osud už dále není předmětem zájmu.

Vývojáři v ERA často potřebují pozorovat, ověřovat a ladit funkci MSS za provozu. Hledá se způsob, jak v laboratorním prostředí nasimulovat realistické vstupní signály a na nich nechat MSS operovat. MSS přijímá zprávy všech formátů používaných leteckými Mode-A, C a S transpondéry. Pro simulace byl zvolen formát nejmodernější a rovněž nejobsáhlejší, tedy ADS-B (Automatic Dependent Surveillance-Broadcast). Cílem této práce je ADS-B zprávy uměle vytvářet a vhodným způsobem je v podobě analogového RF signálu přivádět na anténní vstupy systému MSS. Hlavní překážkou je zde nutnost velmi přesného časování příchodu zpráv. Stejná zpráva musí dorazit na jednotlivé vstupy s přesně nastavitelnými zpožděními.

Coby nástroj, o který se celé řešení opírá, bylo zvoleno populární open-source SDR (Software-Defined Radio) HackRF One. Vývojáři v ERA již HackRF mnoho let používají. Tato platforma skýtá velmi zajímavé specifikace za poměrně nízkou cenu. Předně je však HackRF zcela otevřené po stránce hardwaru i softwaru a lze tak efektivně modifikovat pro speciální účely. Zde je HackRF upraveno způsobem, který umožňuje propojení mnoha jednotek a dosažení přesně synchronizovaného vysílání. Řízené zpožďování ADS-B zpráv je poté dosaženo jednoduše přidáváním nulových vzorků do datového proudu. Kombinace cílených zásahů do hardwaru, do firmwaru pro mikrokontroler i do HDL kódu pro přítomné CPLD má tři hlavní výsledky:

- Propojené jednotky HackRF sdílejí identický hodinový signál pro svůj D-A převodník. Tyto hodiny se shodují jak v kmitočtu, tak zejména ve fázi. Toho bylo dosaženo přivedením externích hodin, dále obejitím některých frekvenčních syntezátorů a nakonec synchronizací hodinových děliček napříč jednotkami.
- Propojené jednotky spouštějí vysílání současně na základě externího signálu. Spouštěcí signál je registrován v CPLD a jelikož dle předchozího bodu běží všechna CPLD na stejných hodinách, je dosaženo prakticky dokonalé současnosti. Spouštěcí signál může být řízen jedním z HackRF, které bylo zvoleno jako master.
- Rozličné buffery na HackRF jsou před každým vysíláním inicializovány validními daty. Díky tomu lze synchonizovaně vysílat opakovaně, bez nutnosti restartu jednotek, cyklování napájení či jiných zvláštních příprav.

Následně se práce zabývá generováním testovacích dat. Úkolem je nasimulovat různý počet letounů (tzv. cílů) pro různý počet pozemních přijímacích stanic systému MSS. Trasa každého cíle je popsána souborem GPX. Prakticky všechny online mapovací služby (`maps.google.com`, `mapy.cz`) umožňují takové trasy vytvářet a exportovat. Dále je třeba přiřadit cíli rychlost a výšku. Tyto parametry jsou v zájmu zjednodušení v rámci

simulace konstantní. Ve standardu ADS-B je cíl rovněž označen šestimístným hexadecimálním ICAO identifikátorem, jenž lze zvolit zcela libovolně. Základnová stanice MSS je poté definována svými koordináty: zeměpisnou šířkou, délkou a nadmořskou výškou. Mnou vytvořená rodina programů `flipe` (Flight Pipeline) postupně tato vstupní data zpracovává za rozsáhlého využití unixového přesměrování výstupů (tzv. piping):

- `flipe-beautify` lze volitelně zařadit pro vyhlazení letové trasy. GPX pocházející z online map se obvykle drží silnic, chodníků, turistických cest, apod. `flipe-beautify` pomocí klouzavého průměrování souřadnic odstraní ostré zatáčky a výsledná trasa tím mnohem lépe připomíná let letadla.
- `flipe-track` přidá časový komponent. Program prochází letovou trasu v zadané výšce, zadanou rychlostí. Ze souřadnic v GPX vybírá pouze ty, na nichž dojde k vyslání ADS-B zprávy, což je standardně každých 500 ms. Kde okamžik vysílání spadá mezi dvojici datových bodů, `flipe-track` vytvoří lineární interpolací bod nový.
- `flipe-adsb` pro každou vysílací pozici cíle syntetizuje věrohodnou ADS-B zprávu coby hexadecimální řetězec. ERA již disponuje interně vyvinutou knihovnou v jazyce Python pro syntézu ADS-B zpráv. Já jsem nevynalézal znovu příslovečné kolo, nýbrž jsem pouze napsal wrapper, který funkce knihovny zpřístupní systému `flipe`.
- `flipe-toa` akceptuje coby argument souřadnice jedné antény MSS. Následně počítá pro každou ADS-B zprávu čas, po který by cestovala k této anténě. Čas cesty je následně připočten k příslušnému času vyslání. `flipe-toa` je nutné zavolat pro každou kombinaci cíle a antény.
- `flipe-mux` přijme výstupy všech instancí `flipe-toa` příslušící stejné anténě a složí je do výstupu jediného tak, aby všechny ADS-B zprávy zůstaly seřazené v čase. Algoritmus velmi blízce přípomíná merge sort. `flipe-mux` se neuplatní, pokud je simulován pouze jeden cíl.
- `flipe-iq` na závěr zakóduje dosud hexadecimální ADS-B zprávy pulzně-poziční modulací (PPM) a převede je na IQ vzorky vhodné pro HackRF. Začátek každé zprávy je umístěn na svůj přesný čas v datovém proudu, prázdný prostor je vyplněn nulovými vzorky. Nutným argumentem je vzorkovací frekvence použitého SDR, od níž se odvíjí jak počet výplňových vzorků, tak počet vzorků na jeden PPM symbol (bitrate pro ADS-B je pevně předepsaný).

Výstup `flipe-iq` je postoupen programu `hackrf_transfer`, jenž je součástí HackRF projektu a zajišťuje komunikaci mezi počítačem a připojeným SDR. Proud IQ vzorků je vyslán na rezervovaném nosném kmitočtu 1090 MHz a koaxiálním kabelem míří přímo do anténního portu přijímacího modulu MSS. Nutno na tomto místě zdůraznit, že v simulaci žádný RF signál necestuje vzduchem. Vysílat umělé ADS-B zprávy do okolí může potenciálně ohrožovat bezpečnost skutečného letového provozu. Neautorizované vysílání na rezervovaném kmitočtu 1090 MHz je trestné.

V první části práce je představen systém MSS a jeho základní funkce, jak se týkají projektu. Druhá část stručně popisuje platformu HackRF One. Třetí část se zabývá samotnou modifikací HackRF pro dosažení žádané synchronizace. Čtvrtá část popisuje algoritmické řešení generátoru ADS-B datových proudů vhodných pro HackRF (a potažmo pro MSS).

DECLARATION

I declare that I have written the Bachelor's Thesis titled "Air Traffic Simulation with HackRF One" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Bachelor's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I would like to thank my colleagues at ERA and consultants of my thesis, Ing. Jan Klusáček and Ing. Jakub Ječábek, for leading me in my engineering pursuits and for patiently answering my numerous questions.

I would also like to thank my academic advisor, Ing. Martin Šťáva, Ph.D., for guiding me through the many pitfalls of formal writing.

Finally, I must not forget to thank my boss at ERA, Ing. František Zouhar. Whatever I needed throughout development - tools, access, or simply more time - he made it happen.

Brno . . . . . .

. . . . . .
author's signature

Contents

List of Figures

Listings

Introduction

Safety in air traffic is an ever ongoing concern. Precise information of aircraft trajectories, speeds, and altitudes is vital to the work of ATC (Air Traffic Control) centres everywhere. Because what good is human expertise without confidence in the data, on which potentially fatal decisions are made.

Today, there are four main families of air surveillance solutions, each of which deals with the challenges of aircraft detection and tracking in a different, unique manner. Usually, these technologies are deployed simultaneously to provide redundancy, data cross-checking, and to cover for each other's shortcomings.

First there is primary radar (PSR). It employs the same principles known since World War II: A radio (RF, Radio Frequency) pulse is transmitted in one direction at a time, an echo is received as the pulse reflects off of metal aircraft bodies. Distance is calculated by measuring the time the signal was travelling. Plane positions are updated every few seconds as the antenna rotates to scan the surrounding airspace. PSR's are large, expensive, and power-hungry. Today, in civil aviation, they continue to be used for their capability to detect non-cooperative targets. That is planes with onboard transponder either broken or switched off deliberately.

Secondary radar (SSR) operates by encoding questions into its RF pulses. Transponders reply to these questions with so-called squawk codes which then get captured by the SSR and processed just as a reflection would. Now the radar signal only has to reach the plane, whereas before it needed to have enough energy to bounce and travel all the way back. Signal strength can be roughly estimated to drop with a square root of distance to find that SSR only requires quarter power compared to PSR. As a result, SSR's are substantially smaller and cheaper to manufacture.

Automatic Dependent Surveillance-Broadcast (ADS-B) technology brings costs even lower while further improving accuracy. ADS-B has planes transmit their complete bundle of tracking information as acquired from GNSS (Global Navigation Satellite Systems), periodically and without any interrogation needed (so-called squitter). Monitoring air traffic becomes as simple as receiving a standardized frequency (1090 MHz) and decoding messages in a standardized format. ADS-B is meant to become the go-to solution for modern air surveillance.

Parallel to the aforementioned technologies, there is a fourth approach, multilateration (MLAT). The same RF message arrives at multiple locations, gets captured, time is noted. There is no finding the total travel time as with radars. Instead, the Time Difference of Arrival (TDOA) between each pair of receiving stations forms a hyperbole on the map (or a hyperboloid in 3D space). The intersection of multiple such hyperboles marks the aircraft position. MLAT can easily coexist with existing systems because it operates on the responses to SSR's interrogation as well as on ADS-B squitter. It does not depend as much on clear lines of sight as radars do, nor does it need to trust data from GNSS.

This thesis revolves around one such MLAT system. Engineers at ERA, who develop the MSS (Multisensor Surveillance System), seek a way to quickly validate

new firmware versions and hardware changes. A solution was proposed to have an SDR (Software-Defined Radio) module connected to each antenna input of the assembled MSS and to simulate the ADS-B messages that would get captured in a real environment. The HackRF One SDR was chosen for this task because it is suitably specced yet inexpensive, and because ERA already has an inventory of these units. Sections 1 and 2 briefly describe the MSS and the HackRF One, respectively.

The first goal of this work is to make modifications to the HackRF and achieve tight synchronization of multiple interconnected HackRF units. This is necessary, because any ADS-B messages, should they translate to the correct location, need to be precisely timed in their arrival to the different MSS inputs. There has to be a level of confidence that whatever is encoded into a digital data stream for the HackRF, also comes out of the RF frontend. And furthermore, that a whole set of carefully crafted data streams does not lose its alignment when sent simultaneously through an array of potentially many HackRF units. Section 3 covers each modification, its purpose, and obtained results.

The second goal is to write a software solution that can generate ADS-B data streams. The input to a flight simulation comprises of a GPX-encoded trajectory (GPS eXchange Format) for each aircraft (each *target*) along with auxiliary parameters such as speed, altitude, and an ICAO identifier (International Civil Aviation Organization). At least three MSS base stations are defined by their fixed coordinates: latitude, longitude, and altitude/elevation. The output is a set of unique data streams, one stream for each MSS station, which carries precisely timed, realistic ADS-B messages. These streams when sent through a HackRF array, fed through a coaxial cable directly into the corresponding MSS antenna ports, shall "play" a complete air scenario for the unsuspecting MLAT system. Section 4 presents how I chose to tackle this programming challenge.

1 Multisensor Surveillance System by ERA

The following section provides a brief overview of the Multisensor Surveillance System. Only the kinds of input that the system receives and the output it returns back shall be of concern here. For all other purposes in this thesis, MSS is a black box, the inner workings of which are both irrelevant to the goals, and also highly confidential.

ERA follows the legacy of Tesla Pardubice, the now-defunct Czech company that pioneered MLAT systems in the world. Research back then used to focus on military applications exclusively and gave its fruits in the form of the rather well-known passive sensors Ramona and Tamara. After the Velvet Revolution of 1989, the state-run Tesla was dissolved and some of their best engineers involved with MLAT systems together founded ERA. A worthy successor to Tamara was developed, called Vera and since superseded by VERA-NG, manufactured to this day. A civil solution has also been developed, aimed to bring MLAT into ATC and in doing so capture a new growing market. So the name MSS was coined. [7]

1.1 Principle of Multilateration

A conventional radar knows both the time it sends an RF signal, and the time a reflection comes back. The direction of the transmission is also a given, as is the location of the radar station. Calculating the position of a reflective object is then straight-forward.

With a passive surveillance system, often incorrectly called "passive radar", incoming RF signals are received, but there is no telling how long those signals have been travelling. However, when receiving the same signals at multiple locations, the Time Difference of Arrival (TDOA) is known 1.1. The following excerpt from ERA's publication [8] explains multilateration without delving into advanced mathematics, while also highlighting the main benefits of MLAT systems in general:

Multilateration employs a number of ground stations, which are placed in strategic locations around an airport, its local terminal area, or a wider area that covers the larger surrounding airspace.

These units listen for "replies", typically to interrogation signals transmitted from a local SSR or a multilateration station. Since individual aircraft will be at different distances from each of the ground stations, their replies will be received by each station at fractionally different times. Using advanced computer processing techniques, these individual time differences allow an aircraft's position to be precisely calculated.

Multilateration requires no additional avionics equipment, as it uses replies from Mode A, C, and S transponders. Furthermore, while the radar and multilateration "targets" on a controller's screen are identical in appearance, the very high update rate of the multilateration-derived targets makes them instantly recognizable by their smooth movement across the screen. A screen displaying multilateration information

can be set to update as fast as every second, compared with the 4 - 12 second position "jumps" of the radar-derived targets.

Assuming just two receiving stations, the set of all points in 3D space for a given TDOA plots a hyperboloid. With three stations, three hyperboloids are obtained and at their intersection, the target. It is worth noting here that while the geometrical notion of a hyperboloid comprises of two sheets, in hyperbolic ranging, only one of the sheets is considered. This is because in addition to the TDOA, we also know which station is *closer* to the target, and so we can rule one of the sheets out. Additional stations serve to enhance accuracy, range, and to provide redundancy.

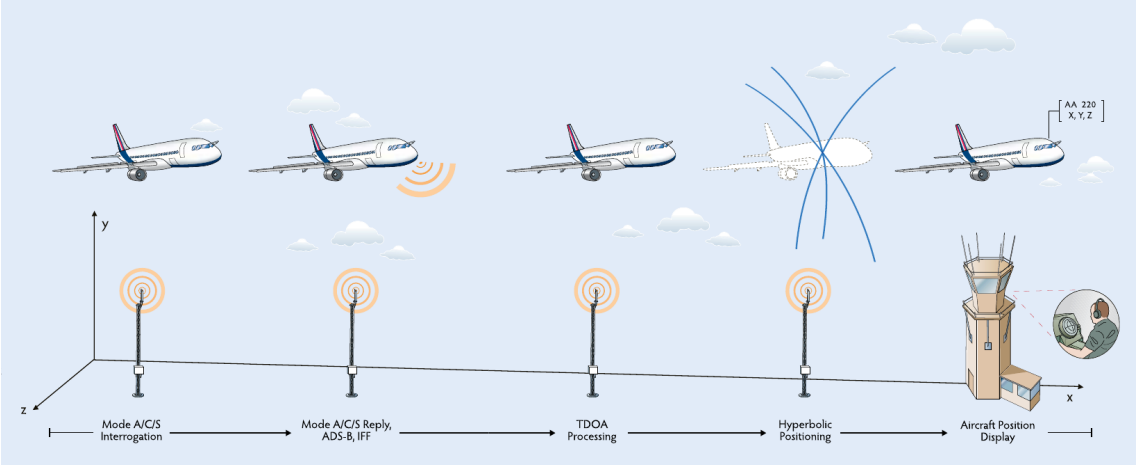

Fig. 1.1: Working principle of an MLAT system, graphic from [8]

1.2 Typical MSS Installation

MSS is a very versatile system. It is well suited for airports, where it enables much higher throughput of the runways. It is also ideal for monitoring the airspace over entire countries, where employing traditional radar would be simply unfeasible without massive blind spots. There is no upper limit to how many base stations can be interlinked and in general, the more the better. Each additional base station provides improved accuracy, especially regarding altitudes, extended range, and redundancy, should any station fail.

In the real world, where price is of great concern, systems get built to the requirements for minimal coverage and minimal accuracy. Advanced computer modelling is used to select optimal locations for the base stations and meet the requirements with the least amount of stations. The base stations send whatever messages they capture along with timestamps to a central server. This is where the actual TDOA processing happens and where a mosaic of the current airspace outlook is assembled. The server is still deployed by ERA and runs proprietary, confidential software. However, it is almost never the case that an air surveillance system would be left to work alone. This is where the ASTERIX standard comes in.

The All Purpose Structured Eurocontrol Surveillance Information Exchange standard is a state-of-the-art data exchange format employed by most modern air surveillance systems. It is designed so that products of many different manufacturers can be interlinked to form a robust solution for airports, national ATC, and similar customers. The standard is developed and maintained by Eurocontrol [9]. MSS central server outputs aircraft tracking information as ASTERIX which is then cross-validated with the already available GNSS tracking information included in ADS-B and with SSR or PSR, if those are deployed. ATC personnel see just a single dot representing each target, but what they are really looking at is the result of taking 2 - 4 such dots and applying clever algorithms to minimize inaccuracies and find the one "true dot". I have attempted to provide a simple illustration in 1.2.

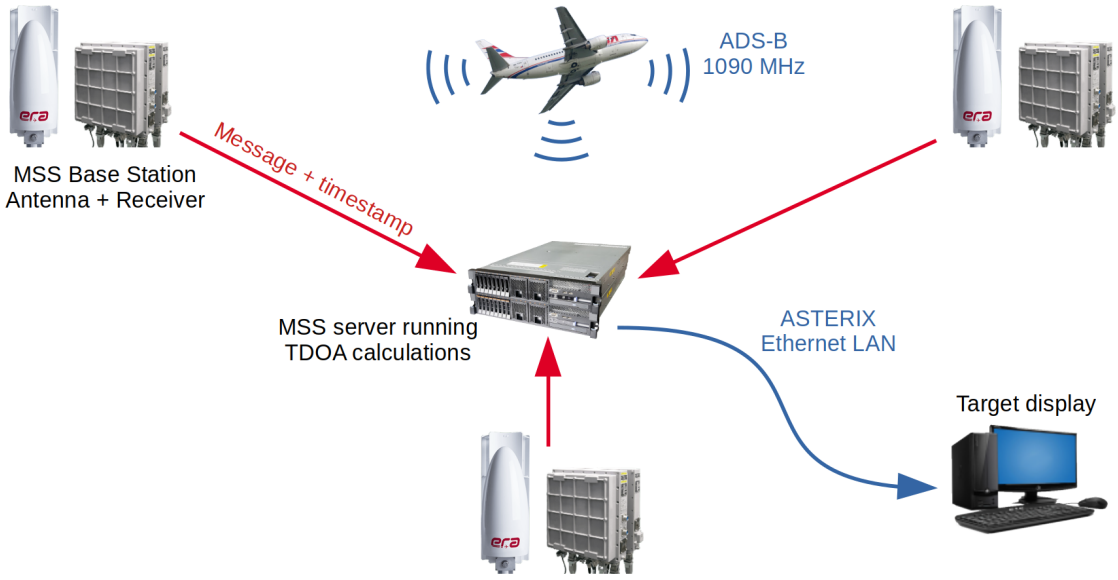

Fig. 1.2: Minimal MSS installation diagram

MSS Test Setup at ERA

In the development workflow at ERA, frequent new changes to MSS firmware need to be tested and validated. For this purpose, a test installation of MSS is available to developers to run their experiments on. Everything is controlled over LAN. In a server rack, there are multiple MSS base units, which would normally be strategically placed in the landscape. With them there is also the central server right beside. Through this server, the entire system can be configured and ASTERIX data extracted. However, having entire MSS assembled in a laboratory is not nearly as useful, as it may seem. From figure 1.3, it should be immediately noticeable why.

Since all antennas share a location, the timestamps end up virtually identical. The system cannot perform multilateration. Or to be precise, it can, with TDOA of zero: any and all targets appear in the circumcenter of whatever locations happen to be configured in MSS's firmware. The circumcenter, if it exists, is the only set of target coordinates with equal distance to all stations.

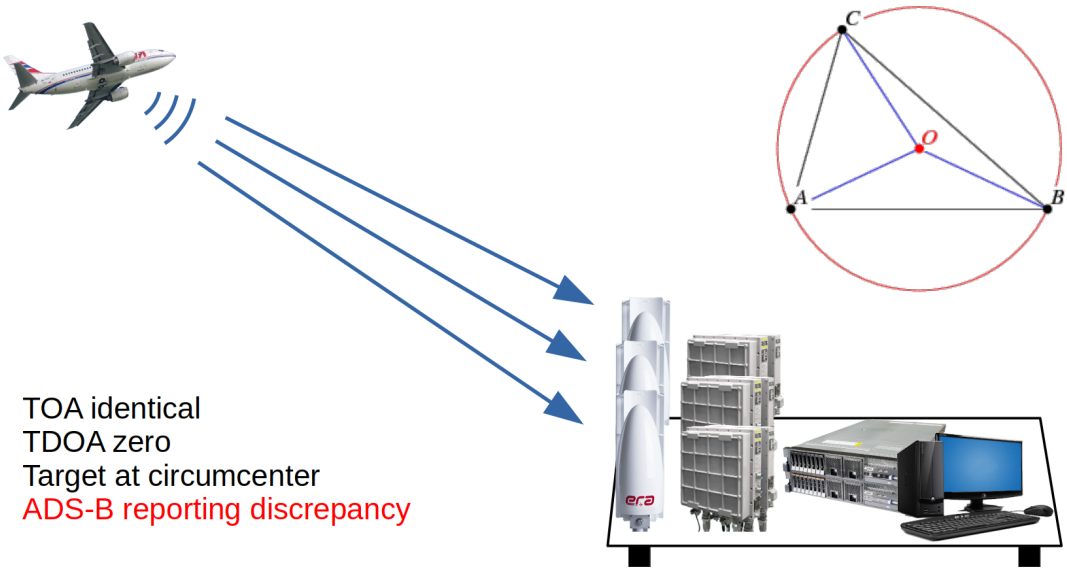

Fig. 1.3: The issue with capturing real data in a lab environment

In addition, since MSS also decodes ADS-B data, and the target position obtained through ADS-B will certainly not be that of the circumcenter, MSS is going to output a generous amount of warning messages due to this cross-referencing discrepancy. Developers are therefore currently constrained to either testing just the decoding of ADS-B, or simulating a single target at the circumcenter using a matching, hand-crafted ADS-B message. This thesis has the potential to extend possibilities substantially.

Each antenna input will be newly served by a distinct HackRF SDR unit, as shown in figure 1.4. Time of arrival will be precisely controlled so that multilateration can be performed on the data. It is highly probable that initial trial runs will be carried out in the minimal configuration of three MSS base stations and three HackRF's. The solution, however, should scale into the future. What we are building is effectively a signal generator with extensible number of channels.

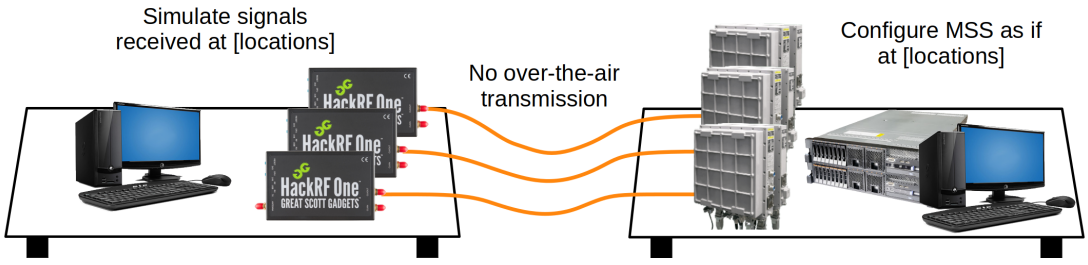

Fig. 1.4: Desired MSS test bench setup

1.3 Overview of the ADS-B Standard

MSS is capable of receiving and decoding all the standard transponder formats used in civil aviation. That is the now-legacy Mode-A and Mode-C, and the modern Mode-S with its various sub-formats. Greatest attention is paid to one of such sub-formats, Automatic Dependent Surveillance-Broadcast (ADS-B), "extended squitter". Excellent concise description courtesy of Garmin:

Under the current Mode-S setup, a standard transponder squit only sends the most basic aircraft identification, system status and pressure altitude information — which ATC's ground computers must correlate with radar tracking information to derive aircraft position, direction of flight, airborne velocity, vertical climb/descent, and so on. Under the new ADS-B concept, each aircraft's approved GPS navigation system will generate all of this data, and then transmit it at least once per second by means of an "extended squitter" — allowing ground controllers and other aircraft in the vicinity to track each airplane's flight path with much greater precision and accuracy. [1]

Structure of all ADS-B messages complies with the specification defined and maintained by ICAO 1.1. They are encoded using Pulse Position Modulation (PPM) and transmitted at the reserved carrier frequency of 1090 MHz. The prescribed bitrate is 1 mega-bit per second. Each extended squit lasts 112 µs. A fixed preamble is added to the beginning of a message to make it easier for a receiver to pick up on. An example of a real ADS-B message is shown in 1.1 and its analog, PPM-encoded form can be seen in 1.5. [29]

Tab. 1.1: Structure of ADS-B messages [29]

nBits	Bits	Abbr.	Name
5	1 - 5	DF	Downlink Format
3	6 - 8	CA	Capability (additional identifier)
24	9 - 32	ICAO	ICAO aircarft address
56	33 - 88	DATA	Data
	[33 - 37]	[TC]	Type Code
24	89 - 112	PI	Parity/Interrogator ID

Listing 1.1: Example of an ADS-B message [29]

```
Raw message in hexadecimal:
8D4840D6202CC371C32CE0576098

[00100]0000010110011
00001101110001110000
110010110011100000

-----+------------+--------------+----------------------+--------------
HEX  | 8D         | 4840D6       | 202CC371C32CE0       | 576098
-----+------------+--------------+----------------------+--------------
BIN  | 10001 101  | 010010000100 | [00100]0000010110011 | 010101110110
     |            | 000011010110 | 00001101110001110000 | 000010011000
     |            |              | 110010110011100000   |
-----+------------+--------------+----------------------+--------------
DEC  | 17    5    |              | [4] ..............   |
-----+------------+--------------+----------------------+--------------
     | DF    CA   | ICAO         | [TC] --- DATA -----  | PI
-----+------------+--------------+----------------------+--------------
```

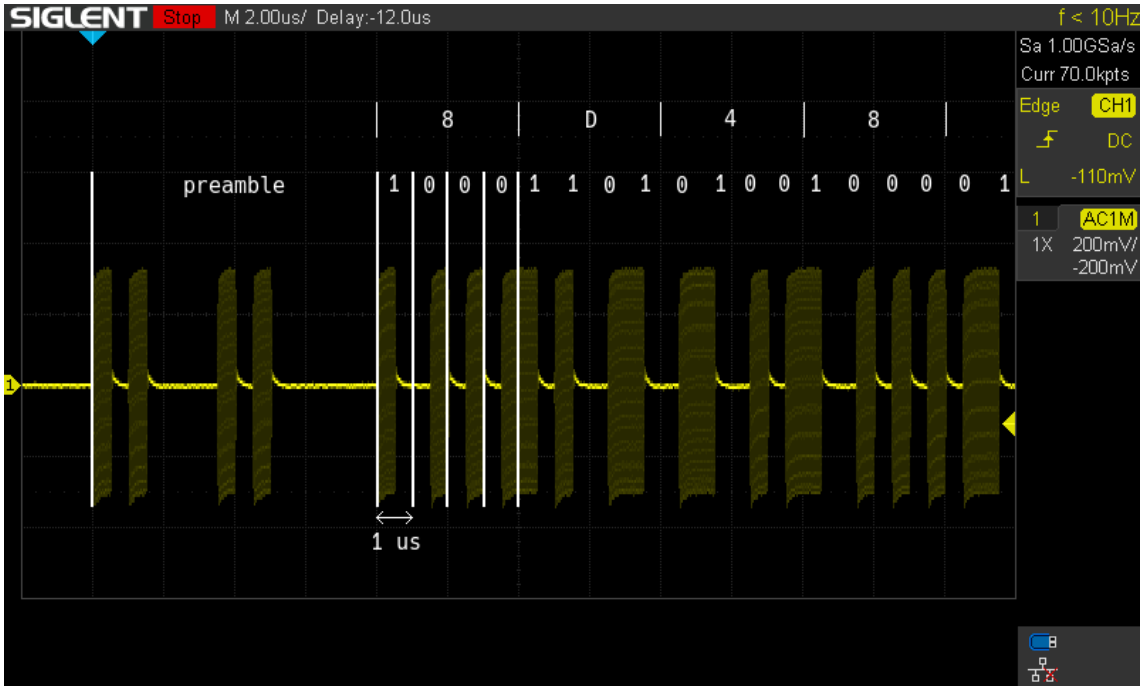

Fig. 1.5: Analog capture of the ADS-B message from 1.1, reduced carrier 80 MHz

2 HackRF One Software-Defined Radio

The following section introduces the HackRF One SDR board [30]. Having been conceived in 2012, with first units shipped in 2014, and being actively improved even today, HackRF has become one of the most popular SDR platforms available. It is fully open-source, open-hardware. Anyone can have their own unit manufactured, although the preferred way of obtaining one is through Great Scott Gadgets [13], the company that serves as a front end for the community's efforts. Principal authorship of the HackRF belongs to Michael Ossmann who himself has contributed the most to the project and its success [1].

What is an SDR

Software-Defined Radio in its broadest sense is, quoting [4] , "radio in which some or all of the physical layer functions are software defined". What this says is that any device which transmits or receives any kind of wireless RF (Radio Frequency) signals and does any kind of digital processing on those signals is in fact an SDR. In 2020, that makes most consumer electronics. A more practical way to think of SDR's is development boards, kits, and dongles that combine digital logic with an RF frontend. This is what people usually mean. A computer or computer peripheral capable of arbitrary RF RX/TX (receive/transmit) operation.

2.1 Specifications and Capabilities

For its modest price among SDR platforms, HackRF offers a rather impressive collection of hardware. Feature list taken from the project wiki [30]:

- Half-duplex transceiver
- Operating frequency: 1 MHz to 6 GHz
- Supported sample rates: 2 MSps to 20 MSps
- Resolution 8 bits (quadrature amplitude modulation)
- Interface: High Speed USB (60 MBps theoretical, 40 MBps accounted for overhead)
- Power supply: USB bus power
- Software-controlled antenna port power (max. 50 mA at 3.3 V)
- SMA female antenna connector (50 ohms)
- SMA female clock input and output for synchronization
- Programmable buttons and pin headers for expansion

HackRF utilizes quadrature modulation with 8 bits of resolution [32]. That means, each sample carries an 8-bit In-Phase (I) component and an 8-bit Quadrature (Q) component, resulting in 16-bit samples. Throughout this thesis, an *IQ-sample* refers to the entire 16 bits. An *I-sample* or *Q-sample* refers to the particular 8-bit component of a complete IQ-sample. When the context is obvious, *sample* may be used to refer to either of the three.

[1] Find Michael Ossmann at <`http://www.ossmann.com/mike/`>

2.2 Hardware Components

In the following description, attention is paid primarily to the TX path as that is the only part of HackRF's functionality relevant to the goals. The ADC/DAC chip may only be referred to as *the DAC*. Data flow will only be considered in the TX direction.

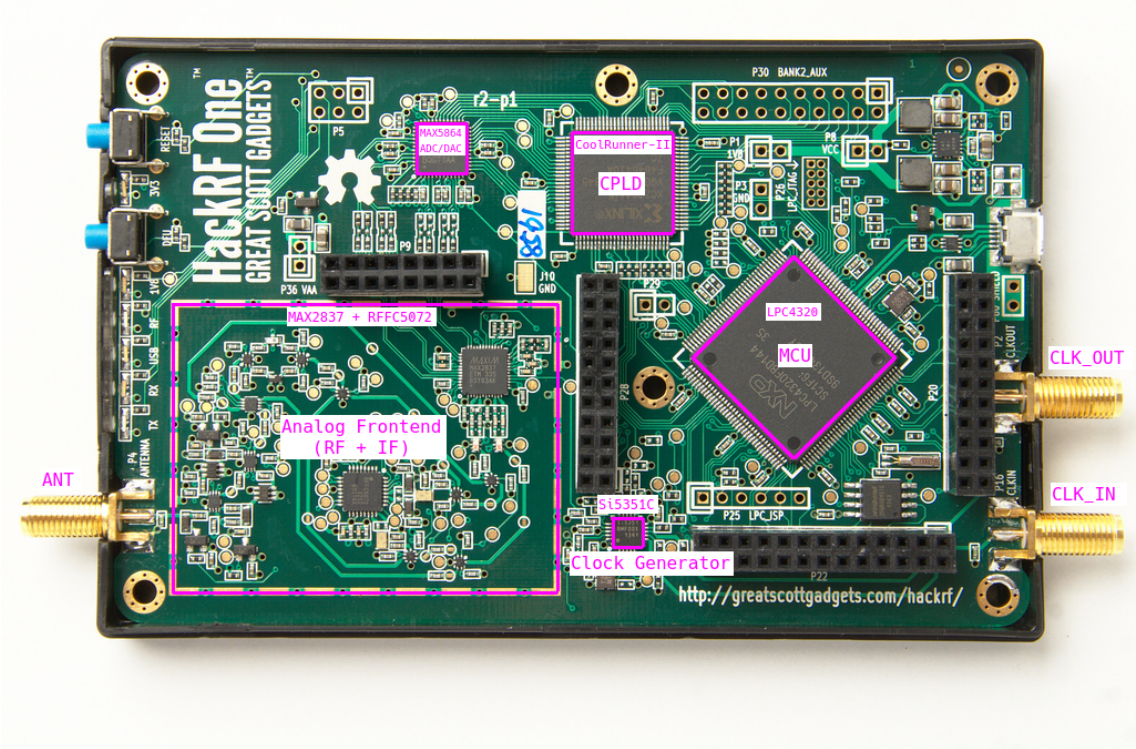

Fig. 2.1: HackRF One select components of interest, photo from [30]

At the heart of HackRF One beats the NXP LPC4320 microcontroller (*the MCU*). It offers one fast ARM Cortex-M4 core and another, slightly slower, Cortex-M0 core (here left unused). The MCU is responsible for all communication with the PC host system. It maintains a USB connection through which IQ-data is received from host. It also manages all the other digital componentry present onboard. IQ-samples leave the MCU through a configurable SGPIO (Serial General Purpose Input-Output) peripheral and enter the CPLD for further processing. [21]

The Xilinx CoolRunner-II CPLD (Complex Programmable Logic Device) serves as glue logic between the MCU and the DAC. The smallest version of a CoolRunner-II is used (XC2C64A [34]) and even then, the part ends up underutilized. This is a feature, not a bug. With more than 50% macrocells still available, 80% after trimming unneeded RX functionality, even ambitious modifications can be implemented, largely unconstrained by resources [35]. More about the CPLD's task in section 2.4.

The Maxim Integrated MAX5864 combined ADC/DAC bridges together the digital and analog domains [19]. It is tailored for radio applications and so features

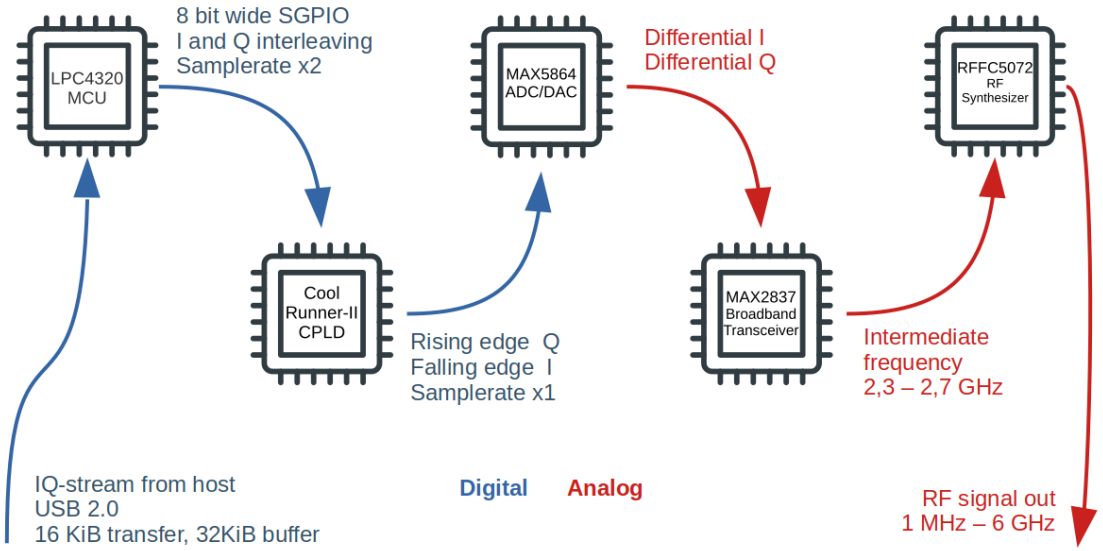

Fig. 2.2: HackRF One data path diagram

dedicated analog differential channels for I-component and for Q-component of a signal. The part is capable of full-duplex operation, 8 bits of RX resolution, 10 bits of TX resolution, and 22 MSps sampling rate. However, due to the maximum speed of the USB 2.0 connection of about 40 MBps [5], only half-duplex mode, 8 bits, and 20 MSps are actually used. The DAC accepts data on both edges of the sample clock (CODEC_CLK); this detail will become important later.

The analog front end can be treated as a monolithic component for the purposes of this work. In reality, it has the Maxim Integrated MAX2837 broadband transceiver [20] working together with the Qorvo (formerly RFMD) RFFC5072 wideband synthesizer/mixer [23]. The MAX2837 takes the analog I and Q components and with them modulates an intermediate frequency (IF). The RFFC5072 is then tasked with up- or down-converting the IF signal to the desired output frequency, set by user. It can be convenient to think of HackRF's architecture as a simple case of Zero-IF, as if the IQ-samples were directly modulating the final carrier, because that is effectively its behaviour. The entire data path through a HackRF, from USB to SMA antenna, is shown in figure 2.2.

The last component worthy of greater interest is the Silicon Labs Si5351C clock generator. It runs off of either the local crystal (CLK_XTAL) or an external clock connected to CLK_IN 2.1. Through a phase-locked loop (PLL) followed by an array of fractional dividers, the Si5351C derives all the various clocks required by individual components. Clocks for the MCU, the MAX2837, and the RFFC5072 remain constant. These chips all use integrated synthesis stages to form their own desired frequencies. Clocks for the DAC, the CPLD, and the SGPIO interface do change and depend on the sample rate requested by user.

2.3 Firmware

All modifications described in this thesis were carried out on the latest firmware release available at time of writing (*v2018.01.1* [2]). However, some changes much in the spirit of this thesis' goals had since been implemented or at least proposed in the master branch [3]. These have been studied for inspiration and backported where appropriate.

Firmware for the HackRF's MCU is written in C. It manages communication with the host PC over USB. It also interfaces with all the configurable onboard components such as the DAC and the chips forming the analog front end. The user can request changes to the sample rate, output frequency and power, configure various filters, etc.

There are the necessary facilities present to allow flashing firmware and writing new bitstreams to the CPLD. Should a firmware update ever fail and leave the device unresponsive ("bricked"), there is a fallback solution in the form of DFU (Device Firmware Upgrade). When a dedicated DFU button is held pressed while power cycling the unit, it enters a special mode and can have known good firmware injected directly into RAM, restoring communication and letting the user try again with an FW flash. [30]

In transmit mode, new data is requested from host periodically in chunks of 16 KiB. There is a 32 KiB static buffer ("the bulk buffer") allocated within the MCU. An ISR (Interrupt Service Routine) is then responsible for reading buffered IQ-samples and sending them downstream through the SGPIO (Serial General Purpose Input-Output) peripheral. More about the data flow and buffering in section 3.3.

The HackRF developers have also attempted to offer some means of multi-unit synchronization out-of-the-box. The user can connect specific header pins with jumper cables and subsequently request simultaneous transmission from multiple units by setting a dedicated flag. The MCU will then continuously poll a pin and only begin streaming IQ-samples once an enable signal is detected on this pin. Such method of synchronization is able to bring down multi-unit time spread to about 50 samples. This can be sufficient for many applications, however, successfully simulating inputs to an MLAT system requires orders of magnitude better precision, the best attainable, in fact. [4]

2.4 HDL Code

The note regarding versioning applies here, same as in section 2.3.

The DAC has a dedicated 8-bit parallel bus for RX and a separate 10-bit bus for TX [19]. The MCU meanwhile uses a single 8-bit bus (the SGPIO) for everything.

[2] <https://github.com/mossmann/hackrf/releases/tag/v2018.01.1>

[3] <https://github.com/mossmann/hackrf/tree/master>

[4] <https://github.com/mossmann/hackrf/wiki/Multiple-device-hardware-level-synchronization>

Listing 2.1: Ensuring correct I- and Q-sample interleaving

```
process (CODEC_X2_CLK)
begin
    if rising_edge(CODEC_X2_CLK) then

        -- I-samples in odd positions
        -- Expected at falling edge of codec clock
        -- Vice versa for Q-samples
        if CODEC_CLK = '1' then

            -- MCU ready to start streaming samples
            if TX_ENABLE = '1' then
                SGPIO_ENABLE <= '1';
            else
                SGPIO_ENABLE <= '0';
            end if;

        end if;
    end if;
end process;
```

HDL code acts as a switch that routes data to the appropriate ADC or DAC pins. Only 8 bits of TX resolution are used, the remaining lower two bits are set to constant '0' in the HDL.

Clock domains get crossed within the CPLD. The MCU rolls out interleaved I and Q samples onto the SGPIO pins on the rising edge of an external clock signal. The DAC accepts data on both edges of its sample clock (CODEC_CLK), however. I-component on the falling edge and Q-component on the rising. The solution here is to have the CPLD operate at double the speed (CODEC_X2_CLK) and maintain correct I/Q interleaving by watching the level of CODEC_CLK (see listing 2.1). [30]

Also, there is a conflict in the encoding of signed byte values: The IQ-data stream from host uses traditional two's complement encoding. The ADC/DAC on the other hand uses a particular implementation of offset binary [19]. HDL serves to translate each byte.

Lastly, in RX mode, HDL takes care of incoming data decimation, should the user demand it. Since the whole of HackRF's RX functionality is irrelevant in this thesis, decimation is only mentioned here for the sake of completeness. This part of HDL has been trimmed out in my modified version.

2.5 PC Host Utilities

HackRF by default is meant to be connected to a host computer and controlled from there. There is a collection of programs distributed as part of the HackRF project

to facilitate this mode of use. These are CLI (Command Line Interface) tools which expose the HackRF's functionality in a human-readable way. For most use case scenarios, they are sufficient. Should they not be, the C library `libhackrf` can be included in custom programs in order to interface with the HackRF at a lower level. [30]

Throughout the work on this thesis, the utilities `hackrf_spiflash` and `hackrf_cpldjtag` have seen extensive use in flashing new code to the MCU and the CPLD respectively. The main tool of interest, however, has to be `hackrf_transfer` which encompasses both the RX and TX capabilities of the HackRF. While RX has not been used once in this case, the TX functions provided by `hackrf_transfer` have proven more than capable and the utility could be used as is without having to dedicate vast amount of time to the study of its inner workings. Section 3.1 will show that a small tweak had to be made in the end, nevertheless.

HackRF host utilities can be compiled to run either on Windows or on a Linux-based operating system. It becomes clear from attempting to use HackRF with the former, however, that Linux is very much the targeted ecosystem. Linux has been used here for the development, for the testing of the modifications made, and is going to be used in the final implementation as well. It is important to mention that some of the tweaks presented herein rely on Linux and its specific implementation of certain libraries, namely `libusb`. Replicating everything while using a Windows host will most likely not yield the same results.

3 Synchronization of Multiple HackRF Units

The following section summarizes the changes made to the HackRF codebase, including the firmware, the host utilities, and the HDL code for the CPLD. Various problems are presented as they were encountered along the way, as well as the steps taken in remedying these problems. A minor but necessary hardware modification is also presented.

3.1 CPLD Level Synchronization

As has been established in the previous section 2.5, in order to even have a chance of being precise enough, any kind of synchronization needs to happen at a low level, within the CPLD. Luckily, a team of researchers from the University of Bologna in Italy and from the European Space Agency, too, have found themselves working with the HackRF and requiring enhanced multi-unit synchronization. Their paper from 2016 explains the HDL tweaks and HW connections necessary to then enable TX from multiple HackRF's simultaneously via an external TTL (Transistor-Transistor Logic) pulse. This was the starting point for my own modifications. [2]

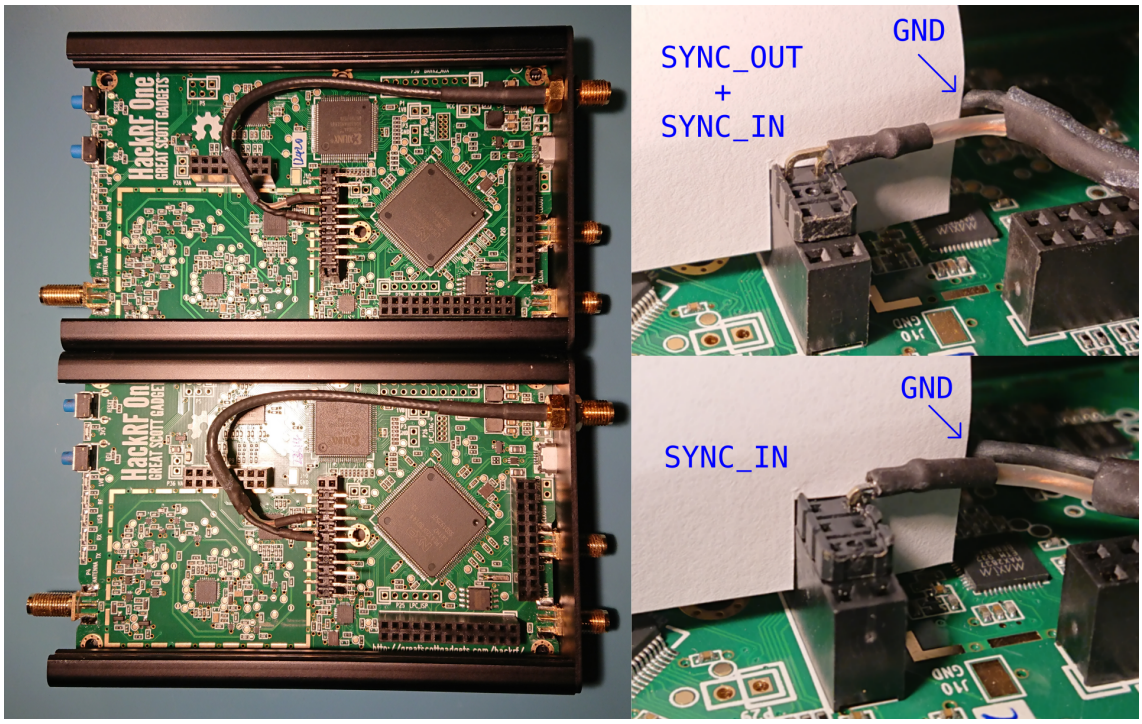

Fig. 3.1: Hardware-modified master unit (top) and slave unit (bottom)

When the PC host requests a transmission, a TX_ENABLE signal tells the CPLD to begin funnelling data to the DAC (see 2.4) . The CPLD then waits for proper alignment of the sample clock, then sets SGPIO_ENABLE high, then begins actually accepting data. This condition can easily be modified to also wait for a synchronization signal to be present. The Italian team used a GNSS time pulse

which they latched for the duration of a transmission. I have encountered issues with edge detection on my units. An incoming pulse would occasionally not register and in some cases, a pulse would get registered that was not there. An exact cause could not be found, but improper termination of the SYNC cable is highly suspected. Nevertheless, I turned to using steady state for the SYNC line and no latching (lst. 3.1). The line is active-low because that way, I can take advantage of pull-up termination mode on the CPLD pins. The Xilinx CoolRunner-II in particular does not support pull-down mode.

The SYNC_OUT signal of each unit stays active during a transmission and would initially relay the state of TX_ENABLE. Later on, I made the MCU control SYNC_OUT instead, the reason for which will be explained further below. The expanded condition remains that SGPIO can only be enabled while SYNC_IN is active. A hardware modification comes into play: all HackRF units have their SYNC_IN lines tied together and a master unit is selected to have SYNC_OUT tied in as well. When a transmission is requested, master effectively enables itself as soon as it is ready and with it the entire SDR array.

Every single HackRF unit must be ready to transmit by the time SYNC_IN arrives. The call to `hackrf_transfer` must have gone through the OS scheduler, data buffers must have been allocated, the hardware signal path configured, etc. A time delay is therefore appropriate before master enables TX operation. However, attempting to synthesize a HDL counter of sufficient width has quickly revealed just how small of a part the CoolRunner-II is. There were simply not enough flip-flops available [35] to create delay longer than a few milliseconds while putting a full stop to any further HDL modifications. Luckily, the complete removal of RX functionality has freed up some GPIO connections to the microcontroller and so the keeping of SYNC_OUT could be offloaded to the MCU entirely, see again listing 3.1 and also 3.2.

PC Host Timeout

In a large array of HackRF's, the slave units will potentially be waiting very long for the SYNC_IN signal. By default, `hackrf_transfer` is programmed to time out after one second of no data sent over USB. With a simple change to the condition, I have allowed the utility to wait alongside a HackRF, indefinitely. A warning message is still printed to `stderr` for every second that passes without successful data transfer, but the connection to the HackRF unit remains open. Should the user ever need to cancel a transmission, the Ctrl+C keyboard shortcut serves to do so.

3.2 Redesigned Clocking Scheme

HackRF comes with a pair of SMA ports, called CLK_IN and CLK_OUT. By default, CLK_OUT drives a weak 10 MHz, 0-3 V square wave clock signal and CLK_IN expects such signal. When a HackRF unit detects this external signal is present, it switches to using it in place of onboard crystal (CLK_XTAL) [30]. The Si5351C clock generator [27] then derives other necessary clocks (fig. 3.2).

Listing 3.1: Implementation of synchronization in HDL code

```
-- SYNC_OUT is now driven entirely by MCU
-- SYNC signal inverted to take advantage of PULLUP
process (CODEC_X2_CLK)
begin
    if rising_edge(CODEC_X2_CLK) then
        SYNC_OUT <= not SYNC_OUT_OVERRIDE;
        i_sync   <= not SYNC_IN;
    end if;
end process;

SYNC_IN_PASSTHROUGH <= i_sync;

process (CODEC_X2_CLK)
begin
    if rising_edge(CODEC_X2_CLK) then

        -- Ensure correct I-Q sample interleaving
        if CODEC_CLK = '1' then

            -- MCU ready and HackRF unit enabled
            if TX_ENABLE = '1' and i_sync = '1' then
                SGPIO_ENABLE <= '1';
            else
                SGPIO_ENABLE <= '0';
            end if;

        end if;
    end if;
end process;
```

Listing 3.2: Implementation of synchronization in firmware

```
// TX enable sequence
if (!already_enabled) {

    // Switch unit to external clock if present
    si5351c_activate_best_clock_source(&clock_gen);

    // Perform synchronization of clock dividers
    do_ext_clk_sync();

    // Pull in half a buffer of valid data
    usb_transfer_schedule_block(
            &usb_bulk_buffer[0x0000],
            0x4000, // 16 KiB
    );
    usb_bulk_buffer_offset = 0;

    // Wait for USB transfers to finish
    delay(150ms);

    // Write valid samples to SGPIO registers
    prime_sgpio_for_tx();

    // MCU ready to transmit
    baseband_streaming_enable(&sgpio_config);

    // Allow other units to become ready
    delay_until_sync(5830000); // 200 ms

    // Trigger transmission (unconnected if slave)
    sync_on();
}
```

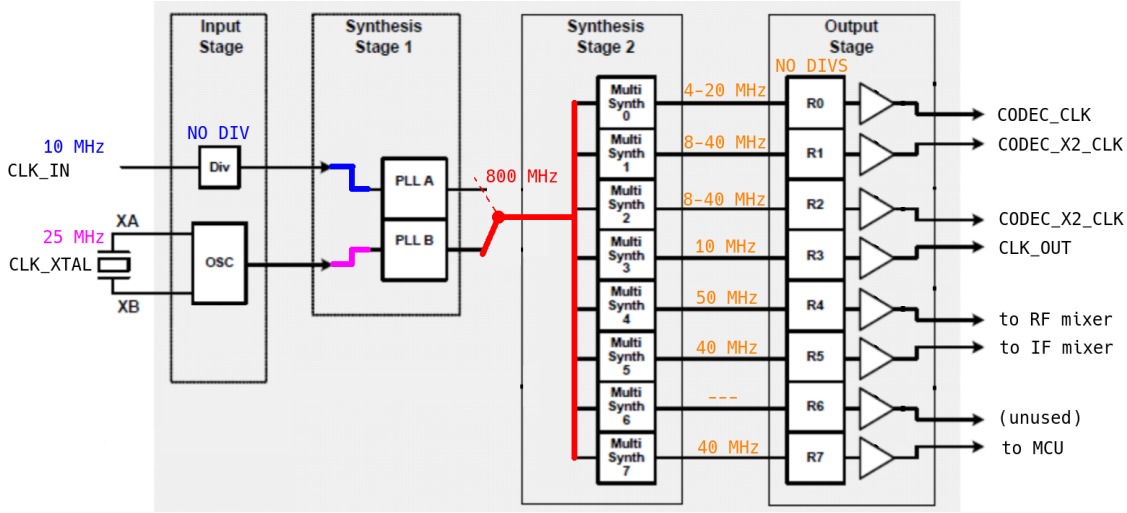

Fig. 3.2: Legacy clocking scheme of a HackRF unit, graphic adapted from [27]

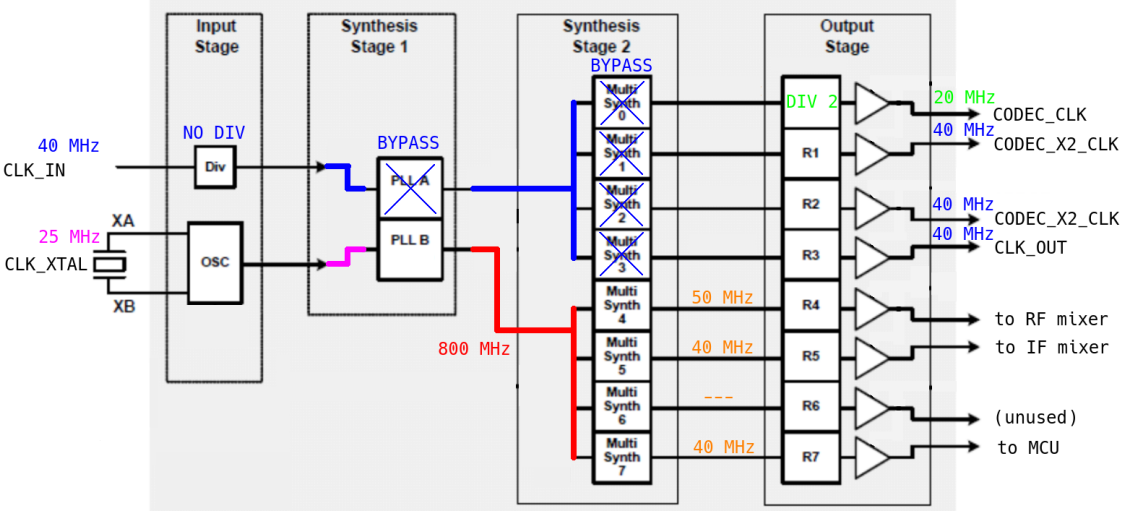

Fig. 3.3: New clocking scheme of a HackRF unit, graphic adapted from [27]

The issue here is that even if all units run off the same precise clock from, say, a stand-alone clock generator, they still go through the process of upconverting to an intermediate clock and then dividing to arrive at local clocks. The Si5351C clock generator introduces a unique phase offset in each unit. The HackRF's end up running at precisely the same frequencies but completely random phases.

This means that each CPLD registers the SYNC_IN signal at a different time. In the worst case scenario, in an array of HackRF's, SYNC_IN might arrive just after a rising edge of CODEC_X2_CLK to one unit, and just before a rising edge to another unit [3]. At the desired sample rate of 20 MSps, this produces a TX start time spread of 50 ns across all "synchronized" units. The researchers in [2] arrived to this very conclusion. I have managed to overcome the issue.

A new clocking scheme was implemented as shown in 3.3. First, the sample

rate had to be chosen and would get hard-coded. A price to pay for improved synchronization. In this case the required sample rate is the highest possible, 20 MSps, 20 MHz for CODEC_CLK. Then, the configuration of the Si5351C was altered with the great help of [28]. CLK_IN is no longer routed to a PLL but rather passed through unchanged to serve as CODEC_X2_CLK and through a simple x2 divider to become CODEC_CLK.

The solution has a caveat in that the x2 divider in each Si5351C may start its division on an odd cycle or on an even cycle of CODEC_X2_CLK. The resulting slower CODEC_CLK clock may end up in exactly opposite phases when compared among units. I have devised a phase synchronization sequence which is run before each transmission and which ensures the same phase of CODEC_CLK in all units.

First, I have implemented another simple x2 divider that is internal to the CPLD. It is held in reset while SYNC_IN is active and operates normally otherwise. With each cycle of the fast CODEC_X2_CLK, the momentary level of this internal divider is compared against the slow CODEC_CLK. A flag is raised on a particular GPIO pin when the two signal levels do not match, that is when CODEC_CLK and the internal div are 180° out of phase. See listing 3.4.

On the MCU side, before a transmission begins, SYNC_OUT gets activated for a short period. Since only master's SYNC_OUT reaches other units, all the internal dividers leave reset state simultaneously and thus become synchronized. Subsequently, the MCU simply reads the aforementioned flag and toggles the inverter of Si5351C's output stage, belonging to CODEC_CLK. This way, all CODEC_CLK clocks become synchronized as well. See listing 3.3

3.3 Data Priming Sequence

Priming of the Bulk Buffer

There is a 32 KiB long buffer ("the bulk buffer") allocated within the MCU. It is logically split in two. Fresh data arrives through USB in chunks of 16 KiB and is deposited into one half then the other, thus forming a textbook example of a ping-pong buffer [33]. Independently, an ISR reads from this buffer 32 bytes at a time and with them replenishes SGPIO shift registers as they funnel data downstream into the CPLD. The ISR keeps track of position in the buffer using a global variable containing current offset from the buffer base address. This variable counts up in increments of 32 and overflows at 32 768 back to zero, reading from the beginning once more.

The default implementation has the pointer into the bulk buffer kept between transmissions. This means, when a new transmission begins after some previous one (without unit reset), the ISR simply takes off wherever it stopped before and it transmits up to an entire half of the bulk buffer, 16 KiB, worth of spurious data. The other half of the bulk buffer is meanwhile scheduled for an USB transfer and contains valid data by the time the ISR gets there.

Listing 3.3: Routine for synchronizing clock dividers across units

```
void do_ext_clk_sync(void)
{
    // Check external clk present
    uint8_t device_status = si5351c_read_single(&clock_gen, 0);
    if (device_status & SI5351C_LOS) {
        return;
    }

    disable_interrupts();

    // Send out a SYNC signal to all CPLDs
    sync_on();
    delay(10ms);
    sync_off(); // This resets dividers

    if (gpio_read(&gpio_clk_out_of_sync)) {
        si5351c_toggle_dac_phase(&clock_gen);
    }

    enable_interrupts();
}
```

Listing 3.4: HDL code for synchronizing clock dividers across units

```
-- Run internal clock divider and compare to onboard clock
-- Reset internal divider via the SYNC line
process (CODEC_X2_CLK)
begin
    if rising_edge(CODEC_X2_CLK) then
        if i_sync = '1' then
            div_q    <= '0';
            badclock <= '0';
        else
            div_q    <= not div_q;
            badclock <= div_q xor CODEC_CLK;
        end if;
    end if;
end process;

CLK_BAD_PHASE <= badclock;
```

I instead make multiple preparations in anticipation of a new transmission. First I set the pointer to 0, that is to the beginning of the bulk buffer, always. I then manually schedule a USB transfer to fill the buffer area right ahead of this pointer. Even the very first interrupt now reads valid data. Please refer back to 3.2.

On a related note, there is a buffer on the host side ("the host buffer"). It is allocated for `hackrf_transfer` by `libusb` when the utility is called. It does not appear to be initialized with valid data and instead causes the HackRF to transmit a substantial amount of zero-samples. This normally does not cause any issues. However, the size of this buffer is platform-dependent (I have found out by using HackRF with different physical computers). In the odd case that multiple HackRF's are synchronized but somehow connected to more than one host, this difference in buffer size may completely throw off the alignment of IQ-streams.

Priming of the SGPIO Peripheral

Once the bulk buffer contains valid data, the last remaining thing to do to ensure a glitch-free transmission is to fill the shift registers of the SGPIO (Serial General Purpose Input-Output) peripheral. The SGPIO is a rather interesting piece of hardware. It consists of up to 16 slices, one such slice is drawn in 3.4. Each slice contains a 32-bit FIFO (First In First Out) register. A single bit can be shifted into the FIFO or out onto an IO pin with each cycle of a dedicated shift clock (here CODEC_X2_CLK). A programmer can configure the slices to operate in parallel and form as wide of a bus as needed, or to daisy-chain the slices and increase register size, or both at the same time. Each slice is double-buffered: one register is being shifted out onto a pin while a second so-called shadow register is presented to the system for read and write operations. Once the data is exhausted, the registers swap and an interrupt is generated. [21]

The HackRF uses 8 simple slices connected in parallel. Such arrangement holds 32 bytes or 16 IQ-samples and shifts out 8-bit I-samples interleaved with 8-bit Q-samples for the CPLD to handle. An ISR (Interrupt Service Routine) periodically reads 32 bytes from the bulk buffer and writes them to the shadow shift register [30]. All of this behaves perfectly fine during normal TX operation. Problems arise at the beginning of a new transmission: The SGPIO contains whatever data was left there from the previous TX call. Additionally, previous operation may have ended with the interrupt flag raised but not serviced. This manifests itself in the form of occasional 800 ns worth of spurious data at the start of a new transmission (16 samples at 20 MSps).

So in order to thoroughly prepare the SGPIO for the start of a new transmission, both the regular and shadow registers need to be filled, and also the interrupt cleared just in case. I have done this by copying the body of the ISR code into a new dedicated function, twice. I have then adjusted addresses used, so that the second copy operates on the actual output register 3.6. As a side effect, a fully primed HackRF unit will have lingering on the SGPIO pins the first I-sample of the data stream. Unless accounted for on the side of the CPLD, this would produce a

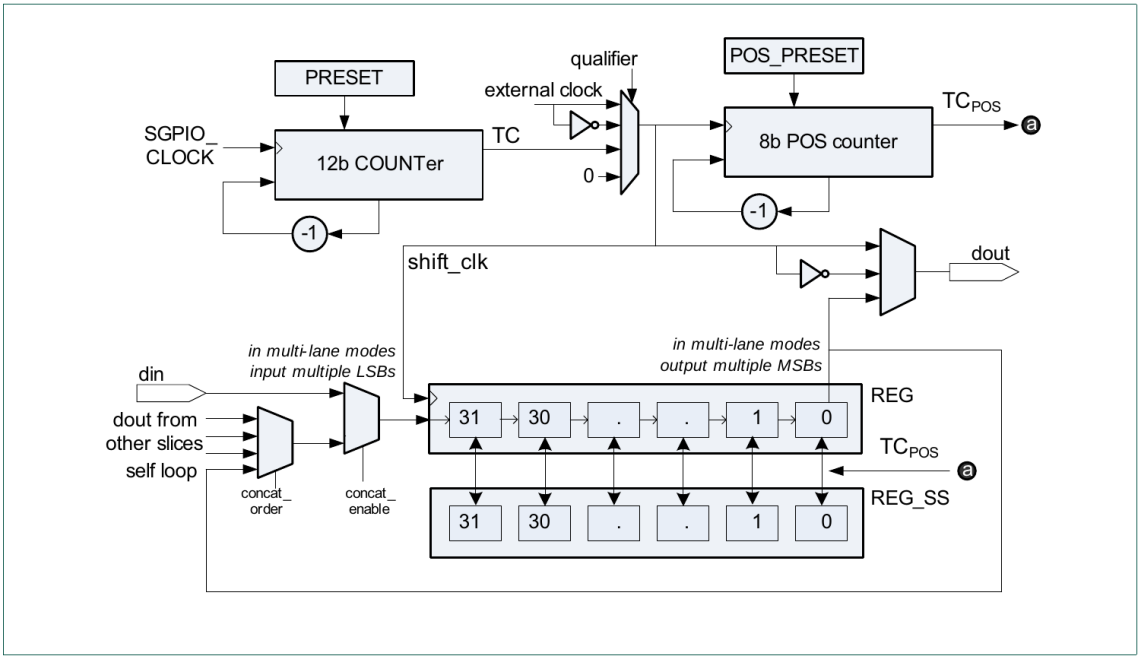

Fig. 3.4: Basic operation of a single LPC4320 SGPIO slice [21]

Listing 3.5: Zero-sample substitution while awaiting enable signal

```
zero_sample <= (others => '0');                                    1
data_from_host_i <= HOST_DATA when SGPIO_ENABLE = '1'              2
                    else zero_sample;                              3
```

continuous spurious signal on the antenna output for the entire time a unit is waiting to be enabled by SYNC_IN. A simple remedy is shown in listing 3.5.

3.4 Usage of a modified HackRF

Multi-unit Clock Interconnection

At least two units need be used for any talks of "synchronization" to be applicable in the first place. There is no known upper limit to how many units can be synchronized. First, as described in 3.2, a common 40 MHz external clock signal must be provided. This clock can originate in one of the units' CLK_OUT, 3.5 shows such situation. Any unit can provide this clock, not necessarily the master. Care must taken when powering on the HackRF array: the clock-providing unit must be powered on last. Alternatively, an external clock source can be used. Again, it must only be enabled after all HackRF units have been powered on. In any case, no matter the final source chosen, a clock signal should always travel the same cable length to each unit so as not to introduce any phase offset. 3.5 shows the master unit looping its CLK_OUT back into itself for this very reason.

Listing 3.6: Filling the SGPIO interface with valid data

```
// Call once when setting up TX
void prime_sgpio()
{
    disable_interrupts();
    clear_sgpio_interrupt();

    int* const p = (int*)&usb_bulk_buffer;

    // Write 64 bytes of valid data to SGPIO
    __asm__(
        "ldr r0, [%[p], #0]\n\t"
        "str r0, [%[SGPIO_REG], #44]\n\t"
        "ldr r0, [%[p], #4]\n\t"
        "str r0, [%[SGPIO_REG], #20]\n\t"

                    ...

        "ldr r0, [%[p], #32]\n\t"
        "str r0, [%[SGPIO_REG_SS], #44]\n\t"
        "ldr r0, [%[p], #36]\n\t"
        "str r0, [%[SGPIO_REG_SS], #20]\n\t"

                    ...

        :
        // Address of the SGPIO shift register
        : [SGPIO_REG] "l" (SGPIO_PORT_BASE + 0x0C0),
        // Address of the SGPIO shadow shift register
        [SGPIO_REG_SS] "l" (SGPIO_PORT_BASE + 0x100),
        [p] "l" (p)
        : "r0"
    );

    usb_bulk_buffer_offset = 64;
    enable_interrupts();
}
```

Multi-unit Synchronization Signal Interconnection

Each unit must be provided with the SYNC_IN TTL signal that stays *high* during setup and by going *low* triggers the transmission. It must also stay *low* for the duration of the transmission. Any free CPLD pin can be configured to accept SYNC_IN. A simple way to provide SYNC_IN that does not require any extra equipment is also shown in 3.5: a selected master unit enables itself as well as any other connected units via the SYNC_OUT signal. The user shall call `hackrf_transfer` for all units roughly simultaneously. The time tolerance can be set in firmware by adjusting the wait time in 3.2.

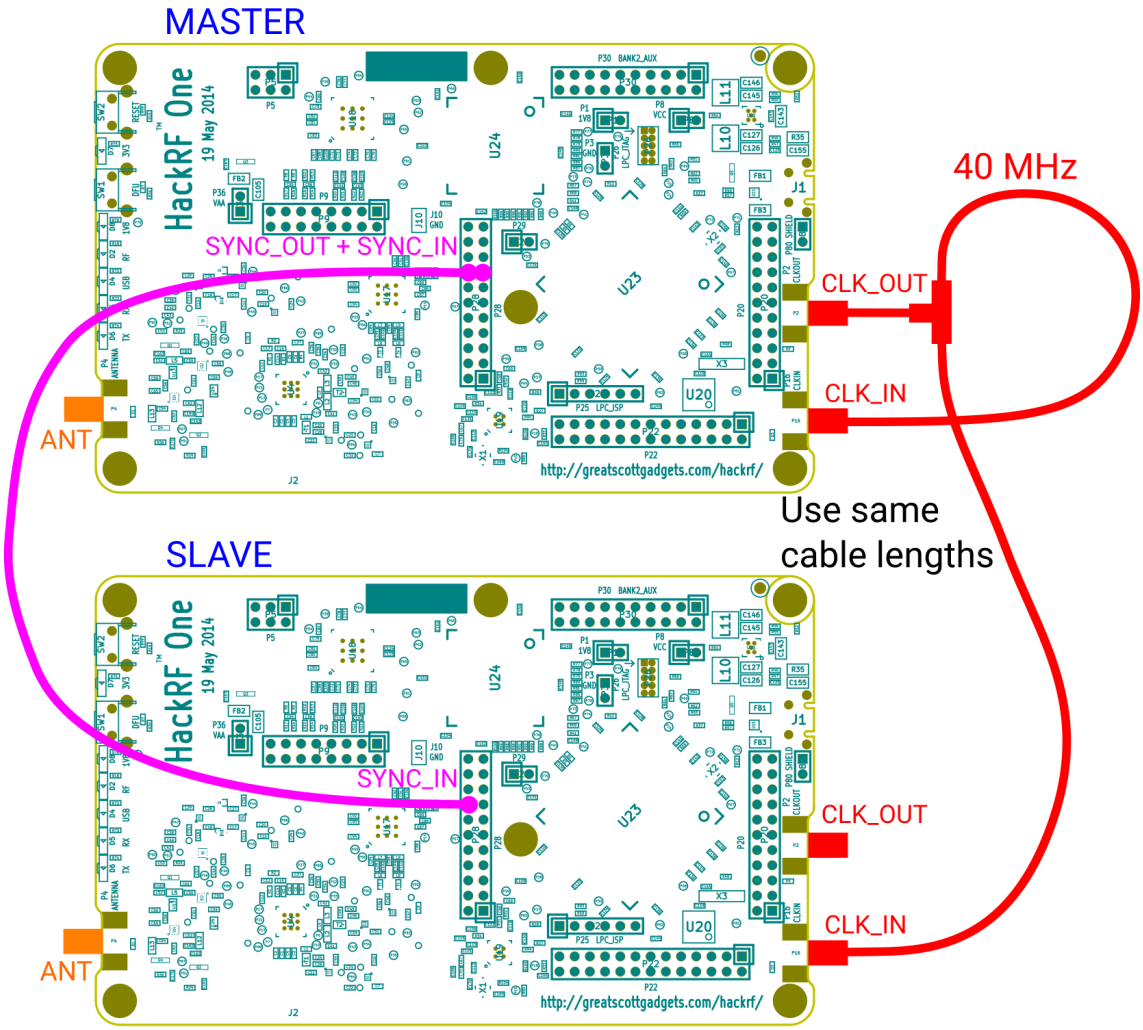

Fig. 3.5: Hardware interconnection of two modified HackRF units, board drawings taken from KiCad project files [30] [22]

Constraints Placed on the IQ-streams

Due to the particular behaviour of `libusb` and by extension `hackrf_transfer`, any IQ-data stream shorter than the host buffer length of 256 KiB will silently fail to

transmit. As a result, no matter how little data the user requires to send, the data stream, be it generated live or saved in a file, should be padded with zeros to reach 256 KiB. In order to stay ahead of `libusb` related bugs, I propose in addition to always keep the data stream a multiple of 256 KiB.

A separate issue, related to the bulk buffer implementation, also requires 32 KiB worth of zeros at the end of an IQ-stream. If this constraint is not met, HackRF will continue cycling through the last 32 KiB of samples in the time between the input stream running out and transmission getting stopped by host. Milliseconds worth of spurious data get transmitted which is both unacceptable and easily prevented. An algorithmic procedure for generating "patched" data streams should be to:

- Write all valid data
- Write 32 KiB worth of zeros
- Round up to nearest 256 KiB multiple by writing more zeros

Showcase of Synchronized Transmission

The following captured transmission shows the kind of precision obtained by modifying HackRF's as described in this section. A pair of HackRF One boards had all of the described modifications made to them, and they were interconnected exactly as shown in 3.5. The same IQ-data test stream was used for both units:

- Only full-amplitude samples (0x7F7F) or zero-samples (0x0000)
- A sequence of three single-sample pulses, also spaced out by single samples (shows that nothing got lost)
- 32 KiB worth of zeros (16 384 samples) at the end of the stream, as prescribed in 3.4
- Full-amplitude samples for the bulk of the stream, to the total size of 256 KiB, again due to 3.4

Each unit was connected to a USB port belonging to a separate root controller of the host PC. Transmissions were requested by calling `hackrf_transfer` from a bash script on a GNU/Linux operating system.

Both radios were transmitting into a 50 Ohm termination which in this case served as a makeshift attenuator. Tee-splitters were used to tap into the signals and the input impedance of the oscilloscope also added some attenuation. Carrier frequency had to be reduced to 80 MHz due to the limitations of the oscilloscope's front end. Channel 1 captures master TX, channel 2 slave TX, channel 3 captures the SYNC_IN signal provided by master.

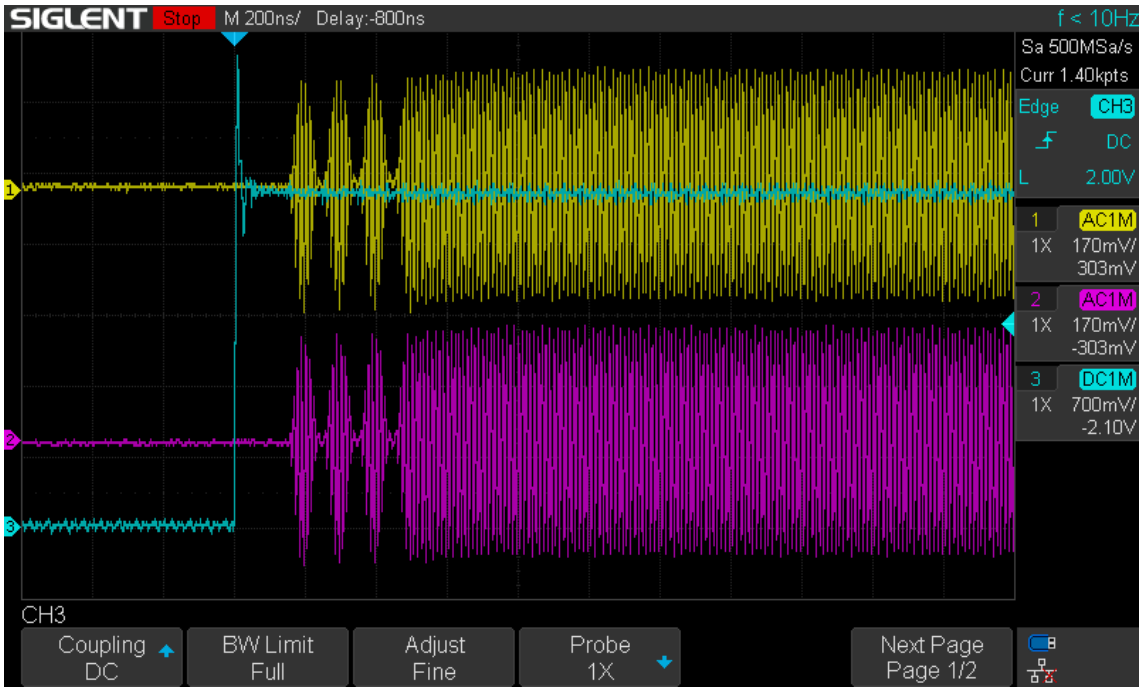

Fig. 3.6: Synchronized TX from two modified HackRF's, master ANT on CH1, slave ANT on CH2, SYNC_IN on CH3, reduced carrier frequency 80 MHz

4 Custom-Generated Transponder Messages

This section deals with the issue of taking test data in a common format and turning it into an IQ-data stream suitable for the HackRF and, by extension, the MSS. This particular part of my work is more of a tech demo rather than a fully fledged, robust solution. The reason being that implementing HackRF flight simulations into the existing test & validation process at ERA carries with it numerous, very specific additional requirements and caveats. Discussing these in the thesis would serve no purpose but cause confusion and detract from the actual point: how to craft MLAT input data. Therefore, I have opted to present a minimalistic approach, completely system-agnostic, and hopefully useful to anyone wanting to create a similar solution for themselves. I try to also mention any areas where my approach is perhaps too minimalistic and could be expanded for further realism.

4.1 Input to the Simulation

First, it is necessary to conceptualize the kind of inputs that a test data generator (which I called `flipe`, short for Flight Pipeline) might take. Engineers at ERA have chosen the GPX (GPS eXchange Format [12]) file format for the encoding of trajectory information. Such file is going to be parsed into a series of discrete GPS coordinates. If more than one plane were to be simulated, then each would have its own GPX file. GPX routes can be obtained through most online mapping providers. I would usually use `https://en.mapy.cz` [18].

Speed is going to be set constant on a per-plane basis. GPX does support encoding speed information alongside each set of coordinates, however, the simulated speed may be required to change run to run. Therefore it has to be a separate input. Similarly for altitude, that too will be configured separately and remain constant throughout a flight. Each plane also gets assigned a made-up ICAO (International Civil Aviation Organization) identifier.

The last remaining input are the coordinates of MSS base stations. These can potentially mimic some existing MSS installations, or be also completely made-up. Both the ADS-B generator and the MSS system must be configured with the same set of coordinates, prior to a simulation run. Without this, the ASTERIX tracking information received back from MSS would be worthless.

4.2 Flipe: The Flight Pipeline

A HackRF unit is connected to each MSS antenna input port. The RF signal leaving each unit is going to carry the same set of ADS-B messages. The difference is in the time that a given message arrives at the various MSS inputs. This time can be adjusted in discrete steps of 50 ns, limited by the sampling rate of the DAC.

The messages may not be in the same order in each stream, even though they usually will be. In a situation where two planes both send their squitter simultaneously,

stations may capture them in either order. The odd case of overlapping messages also needs to be accounted for. Initially, applying a logical OR to the individual overlapping samples may be enough. Artificially producing realistic garbled data is a difficult task that may never be (or may never need to be) solved.

The only differentiating parameter among a set of IQ-streams for a given simulation run is the GPS location of the receiving antennas. This parameter, along with a global speed-of-signal-propagation value, determines the travel time from a plane transmitting to a station receiving. It goes without saying that each IQ-stream must be matched to the corresponding MSS base module, precisely by this set of GPS coordinates.

A collection of UNIX-style [31] programs has been written that when piped into each other turn user input as described above into a set of streams of IQ-samples for as many HackRF units as needed. A modular solution carries the advantage of future adaptability and also allows for better utilization of CPU threads without targeted effort on the side of the programmer [24]. The data flow of my `flipe` family of programs is shown in figure 4.1.

4.3 Descriptions of the Flipe Programs

`flipe-beautify`

This first program in the pipeline is optional. The task of `flipe-beautify` is to create more sensible flight tracks out of tourist routes exported online. I achieved this in one of the most rudimentary ways imaginable: I calculate the moving average of both latitude and longitude separately, and I inject these new values in place of the original ones as I reconstruct the GPX on output. The algorithm I used for calculating new data points is shown in 4.1.

`flipe-track`

This second program in the pipeline parses a GPX file into a simple series of latitude & longitude floating point values. It then applies the given `speed` argument to walk along the flight path and it outputs new data points which now correspond to TX coordinates of ADS-B messages. While the original GPX may have uneven point density, the output points are spaced equally in time. The program applies linear interpolation to achieve this. The time component is added in the format of nanoseconds since simulation beginning, an unsigned 64-bit integer. The distance flown is calculated as great circle distance (see 4.3 and 4.6) using the spherical law of cosines. This is made possible thanks to the constant flight altitude. A constant Earth radius is also assumed. Algorithm shown in 4.2.

`flipe-adsb`

With exact TX coordinates & times known, ADS-B messages can be synthesized. This third program in the pipeline is definitely the most tedious to write from scratch

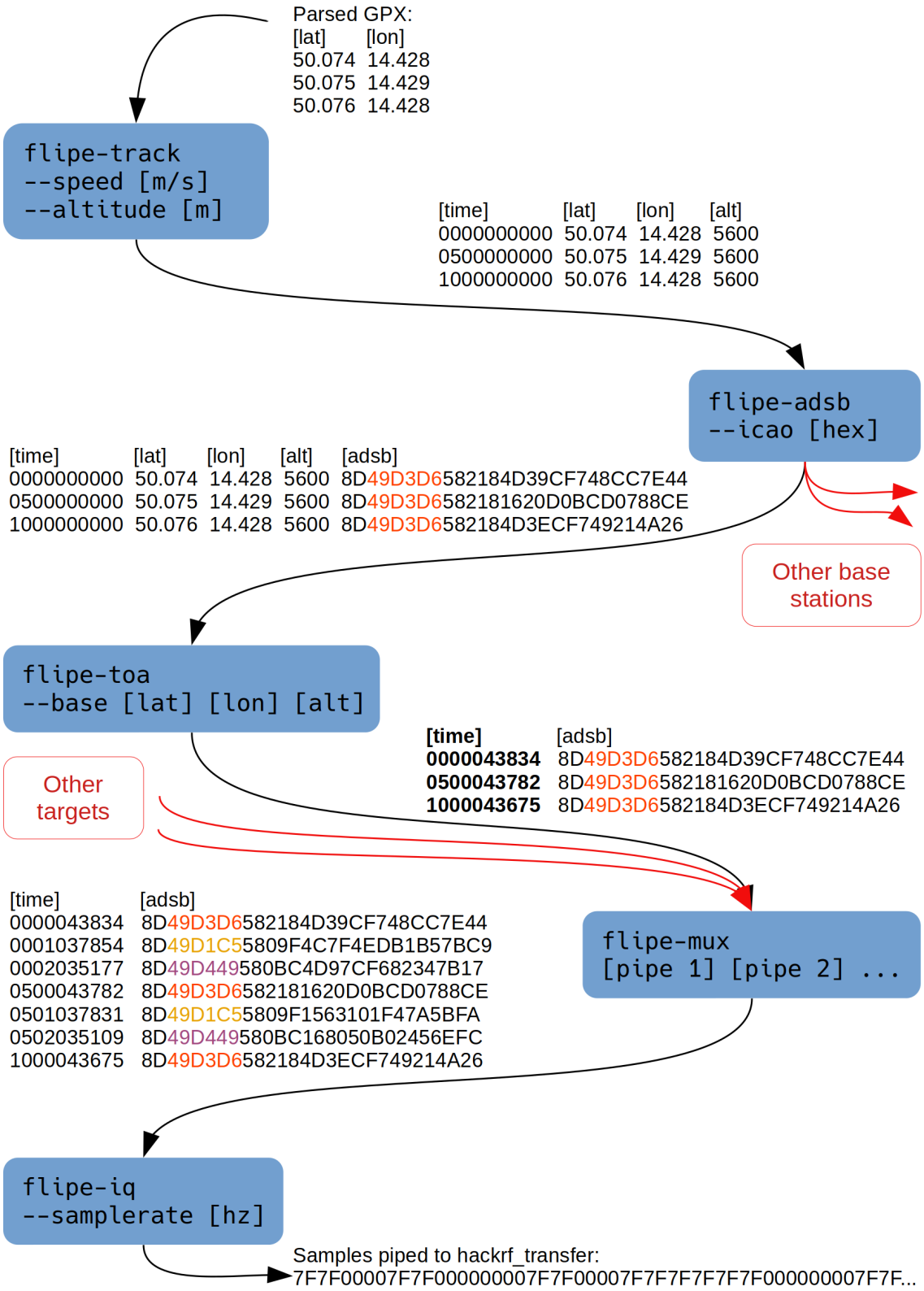

Fig. 4.1: Data flow diagram of the `flipe` suite

Listing 4.1: Function for calculating moving average utilized by `flipe-beautify`

```
#define MA_PERIOD 1000 // Experiment with this setting

void get_moving_average (double *latitude, double *longitude){
    static bool first_run = true;
    static long double lat_sum = 0;
    static long double lon_sum = 0;
    static double lat_buf[MA_PERIOD] = {0};
    static double lon_buf[MA_PERIOD] = {0};
    static int buf_pt = 0;

    // Init code
    // Fill buffers with the initial data point
    if (first_run) {
        first_run = false;
        for (int i = 0; i < MA_PERIOD; i++) {
            lat_buf[i] = *latitude;
            lon_buf[i] = *longitude;
        }
        lat_sum = *latitude * MA_PERIOD;
        lon_sum = *longitude * MA_PERIOD;
        buf_pt = (buf_pt + 1) % MA_PERIOD;
        return;
    }

    // Rotate oldest data point out and new data point in
    lat_sum = lat_sum - lat_buf[buf_pt] + *latitude;
    lon_sum = lon_sum - lon_buf[buf_pt] + *longitude;

    lat_buf[buf_pt] = *latitude;
    lon_buf[buf_pt] = *longitude;
    buf_pt = (buf_pt + 1) % MA_PERIOD;

    // Rewrite the arguments passed by reference
    *latitude = (double) (lat_sum / MA_PERIOD);
    *longitude = (double) (lon_sum / MA_PERIOD);
    return;
}
```

Listing 4.2: Core algorithm of `flipe-track`

```
#define ADSB_RATE 2 // Messages per second
#define NS_IN_S 1000000000 // Nanoseconds per second

// Offset stream beginning by <0;1> TX period at random
// Prevents excessive overlapping from multiple targets
uint64_t time = rand() % (NS_IN_S / ADSB_RATE);

// Distance between ADS-B transmissions
const double step = speed / ADSB_RATE;

elapsed = 0;

// Keep summing segments between GPX data points
while (fetch_datapoint(lat, lon)) {

    dist = get_dist_circle(lat, lon,
            lat_last, lon_last, alt);

    // ADS-B transmission has occured in last segment
    while ((elapsed + dist) > step) {

        // Interpolate a new data point
        coefficient = (step - elapsed) / dist;
        lat_last = lat_last + coefficient * (lat - lat_last);
        lon_last = lon_last + coefficient * (lon - lon_last);

        print(time, lat_last, lon_last, alt);

        time += NS_IN_S / ADSB_RATE; // 0.5s by default
        elapsed = 0;
        dist = get_dist_circle(lat, lon,
                lat_last, lon_last, alt);

        // Loop in case another transmission fits
        // inside the same segment (low probability)
    }

    lat_last = lat;
    lon_last = lon;
    elapsed += dist;
}
```

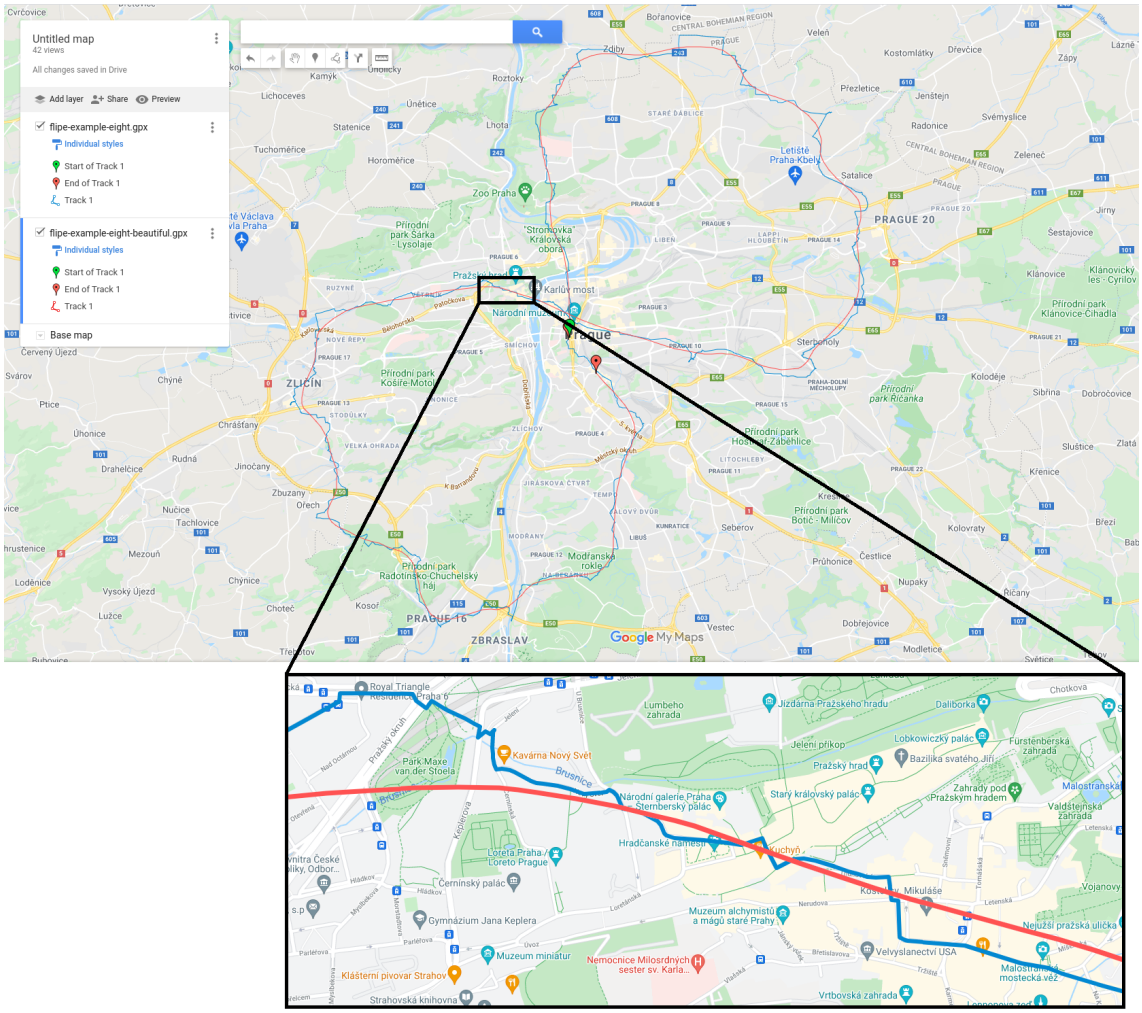

Fig. 4.2: Comparison of GPX tracks with and without `flipe-beautify`

as it needs to adhere to the ICAO specification, should the messages maintain a reasonable level of realism. Fortunately, ERA has already developed an in-house Python library precisely for this task. I merely had to write a wrapper that parses standard input for parameters, feeds these to appropriate classes, and formats the obtained ADS-B hex string on the output. Even though this single program is written in Python and is interpreted rather than compiled, it is perfectly compatible with the rest of `flipe` thanks to pipes.

`flipe-toa`

This fourth program in the pipeline performs the Time of Arrival calculation given the coordinates of a single receiving base station as arguments. Dividing the distance from target to base station by speed of light, a time delta is obtained. This delta can be added to the time of message TX in order to obtain the time of RX of the particular message at that particular base station. Listing 4.4.

The distance to base station is calculated as a straigh line distance, again see 4.3 and 4.3 for code. The altitude of the receiving antenna also has to be provided, and

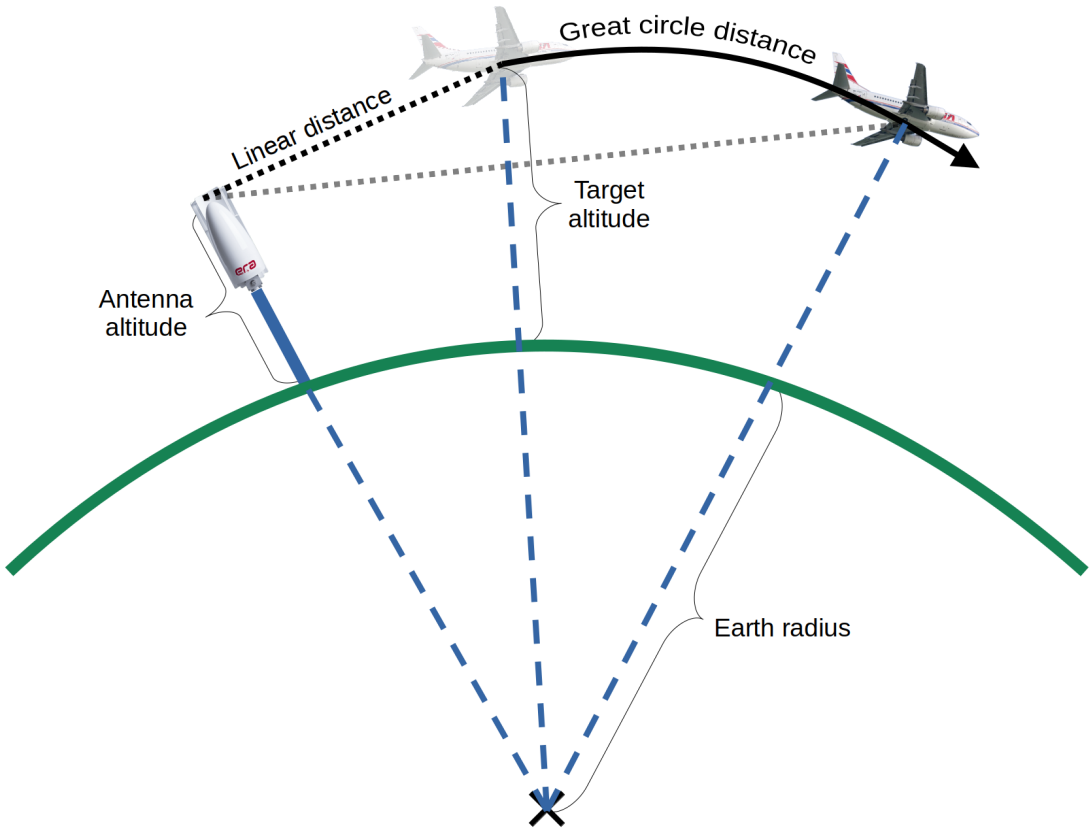

Fig. 4.3: Two types of distances calculated by `flipe`

this particular piece of information is often not easily found in online maps. There are, however, services such as `https://latlongdata.com/elevation/` [16] through which the altitude of a selected location can be obtained. Do not forget to add the height of the mast.

`flipe-mux`

The need for a multiplexer arises from the pipeline style of implementation. There is a separate stream for each combination of target and base station. `flipe-mux` takes all the streams belonging to a single base station as arguments and then performs what strikingly resembles a merge sort. Messages in each stream are already ordered by Time of Arrival and `flipe-mux` merges them into a single stream, keeping the ordering. This stage can be omitted when only one target is being simulated. My particular implementation handles the input streams of varying length using a linked list, see 4.5.

`flipe-iq`

The final processing step produces raw IQ-samples to be sent through a HackRF unit. The ADS-B messages have been represented as hex strings up until this point. `flipe-iq` translates individual bits using PPM encoding and ensures proper time

Listing 4.3: Function to calculate straight line distance

```
double get_dist_line (double lat_a, double lon_a,
                      double lat_b, double lon_b,
                      double alt_a, double alt_b) {

    long double lon_diff = ABS(lon_a - lon_b);
    alt_a = R_EARTH + alt_a;
    alt_b = R_EARTH + alt_b;

    // Spherical law of cosines
    long double cos_phi = (sinl(lat_a) * sinl(lat_b))
        + (cosl(lat_a) * cosl(lat_b) * cosl(lon_diff));

    // Planar law of cosines (SAS triangle)
    return (double) sqrtl(
        (alt_a * alt_a) + (alt_b * alt_b)
          - (2 * alt_a * alt_b * cos_phi) );
}
```

Listing 4.4: The Time of Arrival calculation algorithm

```
// Speed in vacuum / refractive index of air
#define LIGHTSPEED (299792458.0 / 1.000273)
// Nanoseconds per second
#define NS_IN_S 1000000000

while (fetch_datapoint(lat, lon, alt)) {

    double dist = get_dist_line(lat_fix, lon_fix, alt_fix,
                                lat, lon, alt);
    // Time of signal travel in nanoseconds
    uint64_t delta =
        (uint64_t) round((dist * NS_IN_S) / LIGHTSPEED);

    print(time + delta);
}
```

Listing 4.5: Core algorithm of `flipe-mux`

```
while (list_nodes > 0) {

    // Find stream with lowest time value
    // of its next message
    temp = min = linked_list;
    do {
        if (temp->time < min->time) {
            min = temp;
        }
        temp = temp->next;
    } while (temp != linked_list);

    print(min->time, min->msg);

    // Buffer the next data point
    // Unlink stream if exhausted
    if (fetch_datapoint(min, time, msg) == NULL) {

        list_remove_node(min);
        list_nodes--;
    }
}
```

Listing 4.6: Function to calculate great circle distance

```
double get_dist_circle (double lat_a, double lon_a,
                        double lat_b, double lon_b,
                        double alt) {

    long double lon_diff = ABS(lon_a - lon_b);
    alt = R_EARTH + alt;

    // Spherical law of cosines
    long double cos_phi = (sinl(lat_a) * sinl(lat_b))
        + (cosl(lat_a) * cosl(lat_b) * cosl(lon_diff));

    // Get arc length
    return (double) (alt * acosl(cos_phi));
}
```

Listing 4.7: Efficient zero-padding output of `flipe-iq`

```
// Buffer used for write speedup
static const uint16_t zeros[BUFSIZ] = {0x0000};

// Output [padding] amount of zero-samples
for (int i = 0; i < padding / BUFSIZ; i++) {

    // Argument '1' specifies stdin under Linux
    write(1, zeros, sizeof(uint16_t) * BUFSIZ);
}
write(1, zeros, (sizeof(uint16_t) * padding) % BUFSIZ));
```

delays by padding the IQ-stream with zero-samples. A notable part of this program is the optimized loop for writing massive amounts of repetitive data to `stdout`. See listing 4.7. [15] [14]

Conclusion

In this thesis, a new custom solution for automated testing of an air surveillance system has been proposed, researched, and mostly built, save for actual installation at ERA, which was not possible due to the COVID-19 pandemic.

The HackRF SDR has been converted into a specialized tool that transmits ADS-B messages with precise timing, working as one component in a synchronized array of potentially many HackRF units. An improved clocking scheme plays a large role in this. While HackRF supports external clock input out-of-the-box, I have eliminated sources of phase shift that would normally degrade the usefulness of a shared external clock. A great challenge has been to assure that x2 divided versions of this clock do not land in opposite phases among units. A special sequence requiring cooperation from both the CPLD and the MCU has been able to synchronize even these dividers. The SYNC line, described right below, ended up being crucial here, even if not in its originally intended purpose.

Hardware of the HackRF has been slightly altered in the sense that a dedicated CPLD port has been brought out to a new connector, called SYNC_IN. A HackRF unit will now only transmit while SYNC_IN is active. How the signal is provided depends on the use case. Here, from the array of HackRF's a master unit is selected to have another of its CPLD ports, SYNC_OUT, tied directly to SYNC_IN (visualized in 3.5). Subsequently, when all SYNC_IN ports of all units are interconnected, requesting an RF transmission from master enables the entire array. Since the aforementioned modifications have made sure that all units run off the same clock signal, both frequency- and phase-wise, this enabling happens on the very same rising edge, achieving an effectively immeasurable TX start time spread across the array.

Changes have also been made to ensure that no spurious data shall ever be transmitted. This has been achieved by priming with valid data all the various buffers present along the path from the Linux host environment to the SGPIO interface of the LPC4320 microcontroller. Constraints have also had to be set regarding the format of the input IQ-data stream. Certain care must be taken when assembling these files: total size must be a multiple of 256 KiB and the end must be padded with at least 32 KiB worth of zeros. Keeping these guidelines in mind, flawless, repeatable RF transmissions are ensured.

Such synchronized TX operation is a prerequisite for successful air traffic simulation. Now that it has been established, that n-th sample of each data stream arrives at the corresponding MSS antenna input ports simultaneously, delays can be introduced in a controlled fashion through the use of zero-samples. Creating sets of custom-tailored data streams which play a fictional air scenario when sent to the MSS was the second goal.

The Flight Pipeline, or, in short, `flipe` has been created: a family of small programs that can be mixed-and-matched to assemble even complex fictional air

scenarios. Flight tracks are represented by GPX files which can easily be obtained through online mapping providers. Each aircraft (a *target*) has the additional parameters of speed, altitude, and an ICAO identifier (a 6-digit hex string). Each MSS base station has a fixed latitude, longitude, and elevation.

Different `flipe` utilities first process each flight into a series of realistic ADS-B messages along with their time & location of transmission from the onboard transponder. Then, for each pair of target and base station, the travel time of RF signals is calculated and time of reception is obtained. Data belonging to different targets but the same base station is merged into a single stream, which makes the scenario assembly finished. When MSS is configured with the chosen coordinates for its receiving stations and these data streams are transmitted by cable, directly into the corresponding antenna ports, the multilateration should arrive at close to identical flight tracks, speeds, and altitudes.

Even though work on this thesis has been greatly impacted by the pandemic, I have managed to produce an overall concept solution, a ready-to-use firmware, and even a barebones software for building simulations. As I am assured by my colleagues at ERA, the project is suitable for deployment at our department, where it will be used for rapid prototyping, regression tests, and general validation. I consider this thesis work a success.

Bibliography

[1] ADS-B Academy: Extended Squitter. *Garmin* [online]. [cit. 2021-06-03]. Available from: <https://www.garmin.com/en-US/aviation/adsb-squit/>

[2] BARTOLUCCI, Marco, José A. DEL PERAL-ROSADO, Roger ESTATUET-CASTILLO, José A. GARCÍA-MOLINA, Massimo CRISCI and Giovanni E. CORAZZA. *Synchronisation of Low-Cost Open Source SDRs for Navigation Applications* [online]. IEEE, 2016 [cit. 2021-06-03]. 978-1-5090-3885-5/16/$31.00. Available from: <http://spcomnav.uab.es/docs/conferences/Bartolucci_NAVITEC_2016.pdf>

[3] CHU, Pong P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Hoboken, NJ: John Wiley, 2006. ISBN 0-471-72092-5.

[4] COLLINS, Travis F., Robin GETZ, Di PU and Alexander M. WYGLINSKI. *Software-Defined Radio for Engineers.* Norwood, MA: Artech House, 2018. ISBN 978-1-63081-457-1.

[5] COMPAQ, HEWLETT-PACKARD, INTEL, LUCENT, MICROSOFT, NEC, PHILIPS. *Universal Serial Bus Specification* [online]. Rev. 2.0. 2000 [cit. 2021-06-03]. Available from: <https://www.usb.org/document-library/usb-20-specification>

[6] DIE.NET: *Linux Man Pages* [online]. [cit. 2021-6-3]. Available from: <https://linux.die.net/man/>

[7] ERA. *Company history* [online]. [cit. 2021-06-03]. Available from: <https://www.era.aero/en/about-era/history>

[8] ERA. *Multilateration: Executive Reference Guide* [online]. [cit. 2021-06-03]. Available from: <http://www.multilateration.com/>

[9] EUROCONTROL. *All-purpose structured Eurocontrol surveillance information exchange.* [online]. [cit. 2021-06-03]. Available from: <https://www.eurocontrol.int/asterix>

[10] Flightradar24: *Live Flight Tracker* [online]. [cit. 2021-6-3]. Available from: <https://www.flightradar24.com/>

[11] Google Maps [online]. [accessed 2021-6-3]. Available from: <https://www.google.com/maps>

[12] GPX: the GPS Exchange Format. *Topografix* [online]. [cit. 2021-06-03]. Available from: <https://www.topografix.com/gpx.asp>

[13] *Great Scott Gadgets: Open source hardware for innovative people* [online]. [cit. 2021-06-03]. Available from: <https://www.greatscottgadgets.com/>

[14] KERNIGHAN, Brian W. and Dennis M. RITCHIE. *The C Programming Language*. 2nd Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1988. ISBN 0-13-110370-9.

[15] KING, Kim N. *C Programming: A Modern Approach*. 2nd ed. New York: W. W. Norton, 2008. ISBN 978-0-393-97950-3.

[16] Lat Long Data: *Find Elevation* [online]. [cit. 2021-6-3]. Available from: <https://latlongdata.com/elevation/>

[17] LUTZ, Mark. *Learning Python*. 5th ed. Sebastopol: O'Reilly, 2013. ISBN 978-1-449-35573-9.

[18] Mapy.cz [online]. [accessed 2021-6-3]. Available from: <https://en.mapy.cz/>

[19] MAXIM INTEGRATED. *MAX5864: Ultra-Low-Power, High-Dynamic-Performance, 22Msps Analog Front End* [online]. Rev. 1. 2003 [cit. 2021-06-03]. Available from: <https://datasheets.maximintegrated.com/en/ds/MAX5864.pdf>

[20] MAXIM INTEGRATED. *MAX2837: 2.3GHz to 2.7GHz Wireless Broadband RF Transceiver* [online]. Rev. 2. 2015 [cit. 2021-06-03]. Available from: <https://datasheets.maximintegrated.com/en/ds/MAX2837.pdf>

[21] NXP SEMICONDUCTORS. *LPC43xx/LPC43Sxx ARM Cortex-M4/M0 multi-core microcontroller: User Manual* [online]. Rev. 2.1. 2015 [cit. 2021-06-03]. Available from: <https://www.nxp.com/webapp/Download?colCode=UM10503>

[22] OPEN SOURCE. *KiCad EDA Suite* [software]. Ver. 5.1.8, November 2020 [accessed 2021-06-03]. Available from: <https://www.kicad.org/>

[23] QORVO (fmr. RFMD). *RFFC5071/5072: Wideband Synthesizer/VCO With Integrated 6GHz Mixer* [online]. 2014 [cit. 2021-06-03]. Available from: <https://www.qorvo.com/products/d/da000735>

[24] RAMEY, Chet and Brian FOX. *Bash Reference Manual* [online]. Edition 5.1. Free Software Foundation, 2020 [cit. 2021-06-03]. Available from: <https://www.gnu.org/software/bash/manual/bash.pdf>

[25] RASPBERRY PI. *Raspberry Pi Documentation: USB* [online]. [cit. 2021-06-03]. Available from: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/usb/README.md>

[26] RASPBERRY PI. *Raspberry Pi 4 Model B: Datasheet* [online]. 2019 [cit. 2021-06-03]. Available from: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf>

[27] SILICON LABORATORIES. *Si5351A/B/C-B: I2C-Programmable Any-Frequency CMOS Clock Generator + VCXO* [online]. Rev. 1.3. 2020 [cit. 2021-06-03]. Available from: <https://www.silabs.com/documents/public/data-sheets/Si5351-B.pdf>

[28] SILICON LABORATORIES. *Manually Generating an Si5351 Register Map for 10-MSOP and 20-QFN Devices* [online]. Rev. 0.8. 2020 [cit. 2021-06-03]. Available from: <https://www.silabs.com/documents/public/application-notes/AN619.pdf>

[29] SUN, Junzi. *The 1090 Megahertz Riddle: A Guide to Decoding Mode S and ADS-B Signals* [online]. 2nd ed. TU Delft OPEN Publishing, 2020 [cit. 2021-06-03]. Available from: <https://mode-s.org/decode/book-the_1090mhz_riddle-junzi_sun.pdf>

[30] *The HackRF Project Repository* [online]. [cit. 2021-06-03]. Available from: <https://github.com/mossmann/hackrf/>

[31] The Unix Philosophy: A Brief Introduction. *The Linux Information Project* [online]. 2006 [cit. 2021-06-03]. Available from: <http://www.linfo.org/unix_philosophy.html>

[32] Understanding I/Q Signals and Quadrature Modulation. *All About Circuits* [online]. [cit. 2021-06-03]. Available from: <https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/radio-frequency-demodulation/understanding-i-q-signals-and-quadrature-modulation>

[33] WHITE, Elecia. *Making Embedded Systems*. Sebastopol, CA: O´Reilly, c2012. ISBN 978-1-449-30214-6.

[34] XILINX. *XC2C64A CoolRunner-II CPLD* [online]. v2.3. 2008 [cit. 2021-06-03]. Available from: <https://www.xilinx.com/support/documentation/data_sheets/ds311.pdf>

[35] XILINX. *ISE 14.7* [software]. Oct 23, 2013 [accessed 2021-06-03]. Available from: <https://www.xilinx.com/support/download.html>

List of Symbols, Quantities and Abbreviations

ADC	Analog-to-Digital Converter
ADS-B	Automatic Dependent Surveillance-Broadcast
ARM	The CPU architecture developed by Arm Ltd. of the UK
ASTERIX	All Purpose Structured Eurocontrol Surveillance Information Exchange
ATC	Air Traffic Control
C	The programming language "C"
CLI	Command Line Interface
CLK	Clock (reg. digital electronic circuits)
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DFU	Device Firmware Upgrade
DSP	Digital Signal Processing
ERA	The Czech company ERA, a.s.
FIFO	First In First Out
FW	Firmware
GCC	The GNU C Compiler
GNSS	Global Navigation Satellite Systems
GNU	The GNU Project ("GNU is Not Unix")
GPIO	General Purpose Input-Output
GPS	Global Positioning System
GPX	GPS eXchange Format
HDL	Hardware Description Language
HW	Hardware
ICAO	International Civil Aviation Organization
IO	Input-Output

IQ	In-Phase, Quadrature
ISR	Interrupt Service Routine
LAN	Local Area Network
MCU	Microcontroller Unit
MLAT	Multilateration
MSS	Multisensor Surveillance System
PC	Personal Computer
PLL	Phase-Locked Loop
PPM	Pulse Position Modulation
PSR	Primary Surveillance Radar
QAM	Quadrature Amplitude Modulation
RF	Radio Frequency
RTL	Register-Transfer Level
RX	Receive/Receiver/Reception
SDR	Software-Defined Radio
SGPIO	Serial General Purpose Input-Output
SMA	SubMiniature version A (connector type)
SSR	Secondary Surveillance Radar
SW	Software
SYNC	Synchronization
TDOA	Time Difference of Arrival
TOA	Time of Arrival
TTL	Transistor-Transistor Logic
TX	Transmit/Transmitter/Transmission
USB	Universal Serial Bus
VHDL	The VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XTAL	Crystal (reg. clock signals and oscillators)

MSps Mega-Samples Per Second

MBps Mega-Bytes Per Second

---

1 byte (B) = 8 bits (b)
1 KiB = 1024 B
1 MiB = 1024 KiB
1 (IQ-)sample = 16 b
1 I-sample = 1 Q-sample = 8 b

CLK_XTAL Frequency of the local crystal, only used as input to the Si5351C clock generator, 25 Mhz

CLK_OUT Clock signal output present on every HackRF unit, 10 MHz by default, 40 MHz modified

CLK_IN Clock signal input present on every HackRF unit, expected 10 MHz by default, 40 MHz modified

CODEC_CLK The clock used by the MAX5864 ADC/DAC, equal to sample rate

CODEC_X2_CLK The clock of the CoolRunner-II CPLD and also of the SGPIO interface, double the sample rate

SYNC_OUT TTL output signal of each HackRF unit, held high while transmitting

SYNC_IN TTL input signal to each HackRF unit, must stay high in order to transmit

TX_ENABLE Signal telling the CPLD to begin streaming data

SGPIO_ENABLE Signal from CPLD used to enable the MCU's SGPIO interface

bulk buffer A 32 KiB ping-pong style buffer within the MCU

host buffer A 256 KiB buffer allocated on PC host side by `libusb`

shift register Two 32 B registers move data through SGPIO in a double-buffered configuration

A Example Simulated Air Scenario

This appendix demonstrates the intended usage of the `flipe` toolkit. Three targets are placed above Prague, the Czech capital, and three vantage points are chosen for the base stations. Auxiliary parameters of the targets belong to actual aircraft that could be observed above Czechia at time of writing using FlightRadar24 [10]. The associated tracks are purely fictional, however.

Figures A.1 , A.2 , and A.3 show the tracks being exported from an online mapping provider, Mapy.cz [18] . Figure A.4 shows the locations for MSS base stations being selected in a similar way. After being processed by `flipe-beautify` with the moving average period set to 100, the tracks have been uploaded again, this time to Google Maps [11], to visualize the entire scenario A.5.

Listings A.1 and A.2 together form a script that could be run on a Linux system and which would call all the `flipe` utilities in order, and ultimately transmit the resulting IQ-streams through three HackRF's. With no good option to showcase the actual transmissions here, I have at least included the last human-readable stage of test data in listing A.5 , A.3 , and A.4 . The two columns represent time of reception at an antenna expressed as nanoseconds since simulation start, and the ADS-B message expressed as a hexadecimal string.

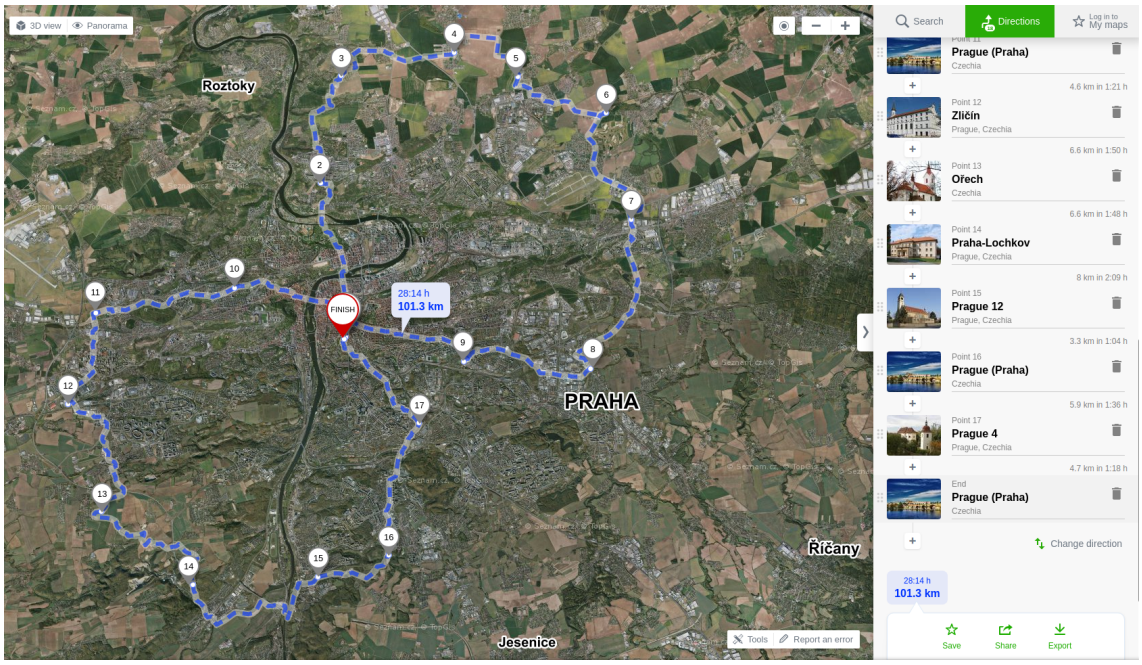

Fig. A.1: Drawing the first eight-shaped flight track, Mapy.cz [18]

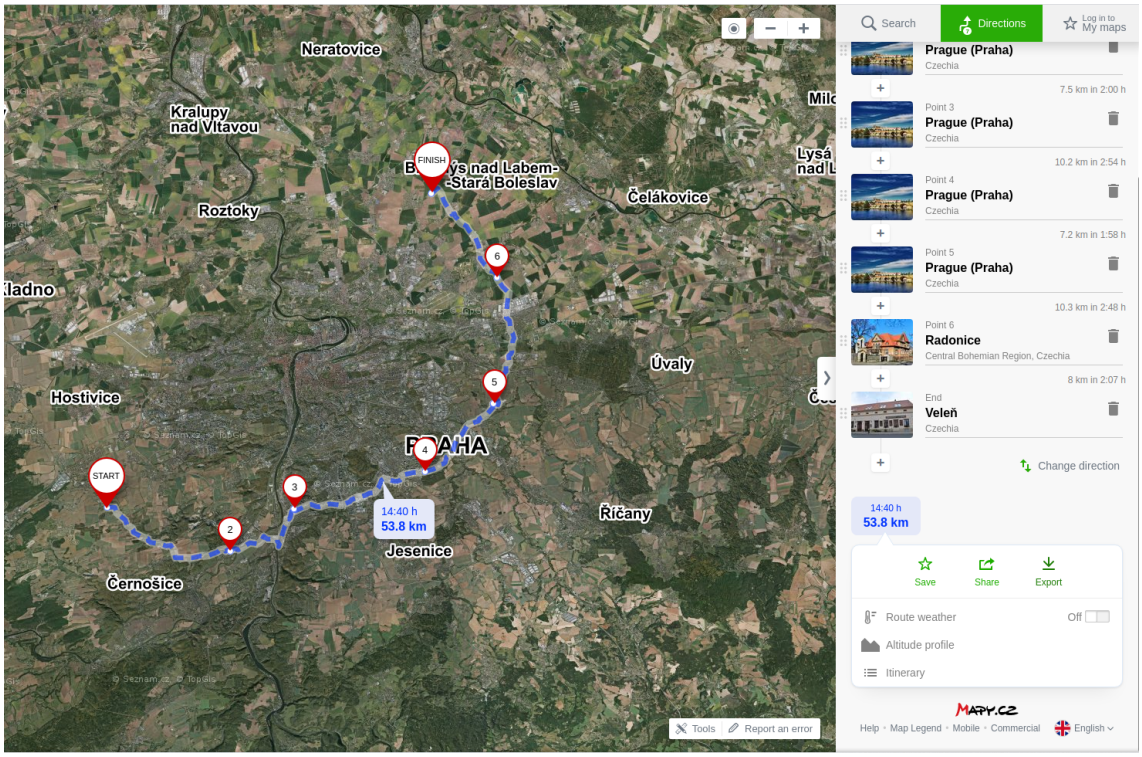

Fig. A.2: Drawing the second curve-shaped flight track, Mapy.cz [18]

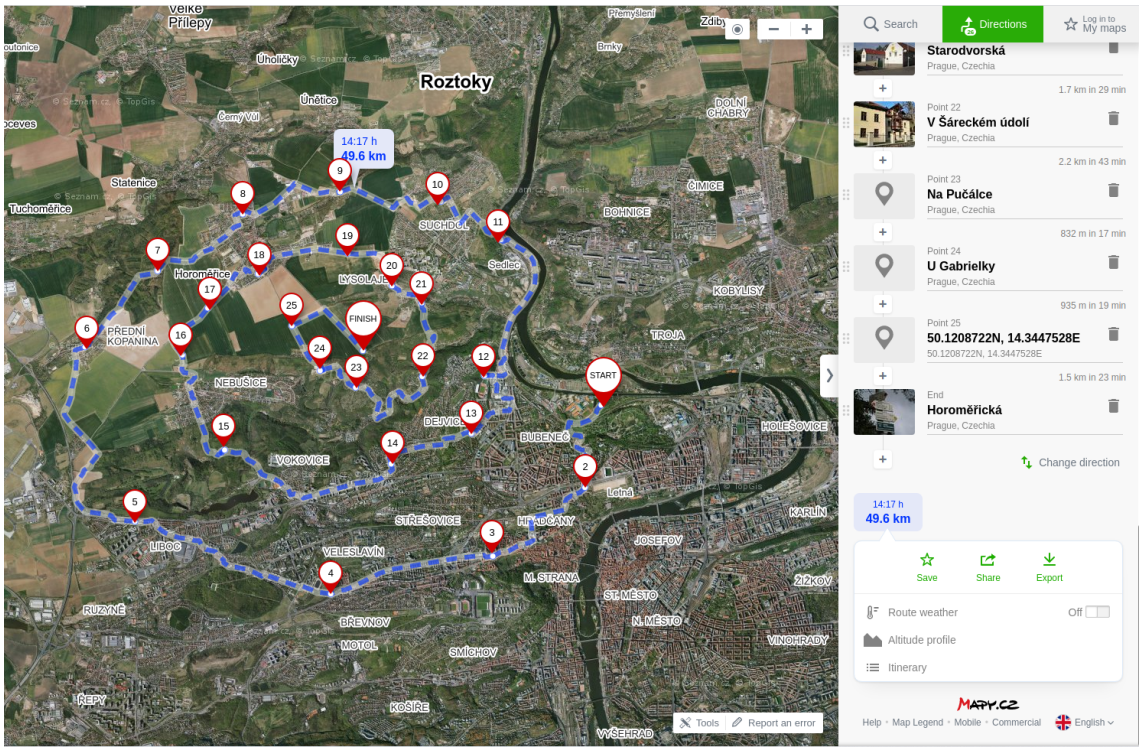

Fig. A.3: Drawing the third spiral-shaped flight track, Mapy.cz [18]

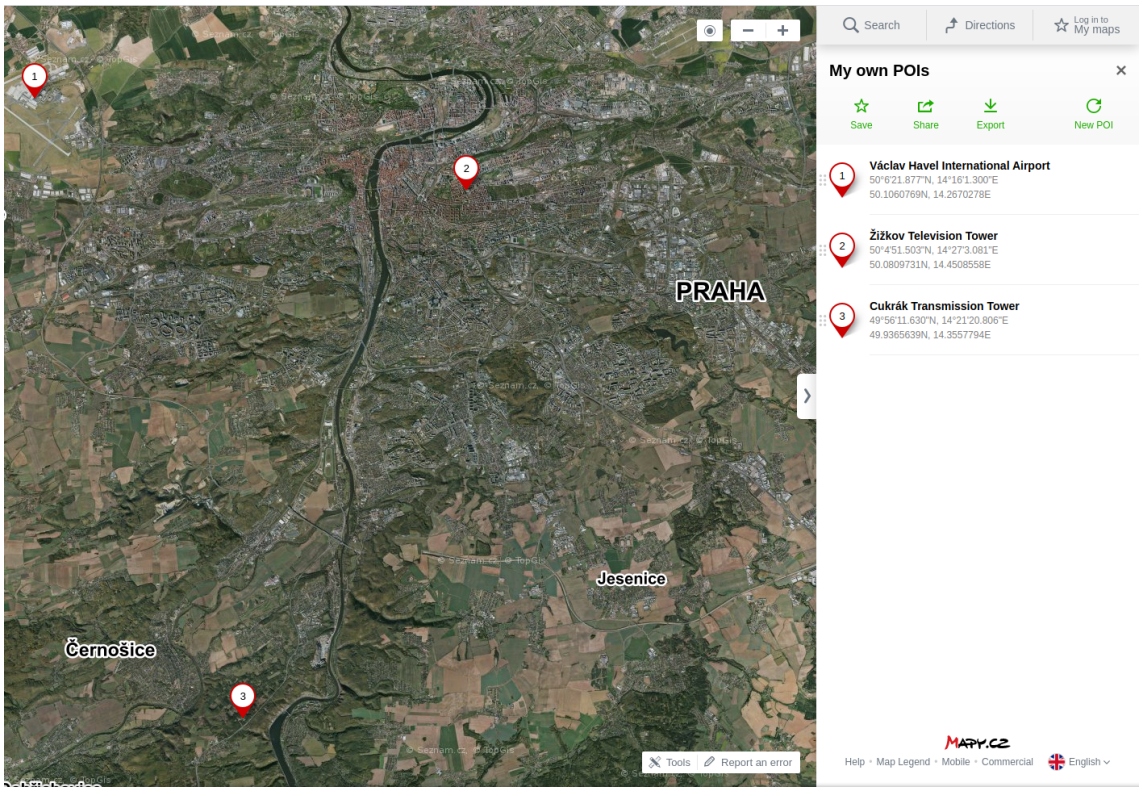

Fig. A.4: Choosing locations for three MSS base stations, Mapy.cz [18]

Listing A.1: Scripting a scenario in Bash, parameter declarations

```
#!/bin/bash

# HackRF hardware UUIDs
ID_ALPHA="00000000000000001111111111111111" #master
ID_BRAVO="00000000000000002222222222222222"
ID_CHARLIE="00000000000000003333333333333333"

# Base station coordinates
GPS_ALPHA="50.1060769␣14.2670278␣426"
GPS_BRAVO="50.0809731␣14.4508558␣474"
GPS_CHARLIE="49.9365639␣14.3557794␣594"

# Target params
TRK_EIGHT="example-eight.gpx"
ICAO_EIGHT="49D3D6"
SPD_EIGHT="206"
ALT_EIGHT="5600"

TRK_CURVE="example-curve.gpx"
ICAO_CURVE="49D1C5"
SPD_CURVE="67"
ALT_CURVE="965"

TRK_SPIRAL="example-spiral.gpx"
ICAO_SPIRAL="49D449"
SPD_SPIRAL="65"
ALT_SPIRAL="1300"

# HackRF params
FC="1090000000"  # Carrier frequency 1090 MHz
FS="20000000"    # Sample rate 20 MSps
BW="28000000"    # Filter bandwidth (maximum for TX)

# Create named pipes
mkfifo ALPHA-EIGHT
mkfifo ALPHA-CURVE
mkfifo ALPHA-SPIRAL
mkfifo BRAVO-EIGHT
mkfifo BRAVO-CURVE
mkfifo BRAVO-SPIRAL
mkfifo CHARLIE-EIGHT
mkfifo CHARLIE-CURVE
mkfifo CHARLIE-SPIRAL
```

Listing A.2: Scripting a scenario in Bash, calling `flipe` utilities

```
# Process individual flights
cat $TRK_EIGHT | flipe-beautify |
flipe-track $SPD_EIGHT $ALT_EIGHT |
flipe-adsb.py $ICAO_EIGHT > EIGHT.intermediate &

cat $TRK_CURVE | flipe-beautify |
flipe-track $SPD_CURVE $ALT_CURVE |
flipe-adsb.py $ICAO_CURVE > CURVE.intermediate &

cat $TRK_SPIRAL | flipe-beautify |
flipe-track $SPD_SPIRAL $ALT_SPIRAL |
flipe-adsb.py $ICAO_SPIRAL > SPIRAL.intermediate &

wait

# Process each flight+station pair
cat EIGHT.intermediate | flipe-toa $GPS_ALPHA > ALPHA-EIGHT &
cat CURVE.intermediate | flipe-toa $GPS_ALPHA > ALPHA-CURVE &
cat SPIRAL.intermediate |
flipe-toa $GPS_ALPHA > ALPHA-SPIRAL &

cat EIGHT.intermediate | flipe-toa $GPS_BRAVO > BRAVO-EIGHT &
cat CURVE.intermediate | flipe-toa $GPS_BRAVO > BRAVO-CURVE &
cat SPIRAL.intermediate |
flipe-toa $GPS_BRAVO > BRAVO-SPIRAL &

cat EIGHT.intermediate |
flipe-toa $GPS_CHARLIE > CHARLIE-EIGHT &
cat CURVE.intermediate |
flipe-toa $GPS_CHARLIE > CHARLIE-CURVE &
cat SPIRAL.intermediate |
flipe-toa $GPS_CHARLIE > CHARLIE-SPIRAL &

# Merge results and transmit
flipe-mux ALPHA-EIGHT ALPHA-CURVE ALPHA-SPIRAL |
flipe-iq $FS | hackrf_transfer -d $ID_ALPHA
                -f $FC -s $FS -x 47 -a 0 -b $BW -t - &

flipe-mux BRAVO-EIGHT BRAVO-CURVE BRAVO-SPIRAL |
flipe-iq $FS | hackrf_transfer -d $ID_BRAVO
                -f $FC -s $FS -x 47 -a 0 -b $BW -t - &

flipe-mux CHARLIE-EIGHT CHARLIE-CURVE CHARLIE-SPIRAL |
flipe-iq $FS | hackrf_transfer -d $ID_CHARLIE
                -f $FC -s $FS -x 47 -a 0 -b $BW -t - &
```

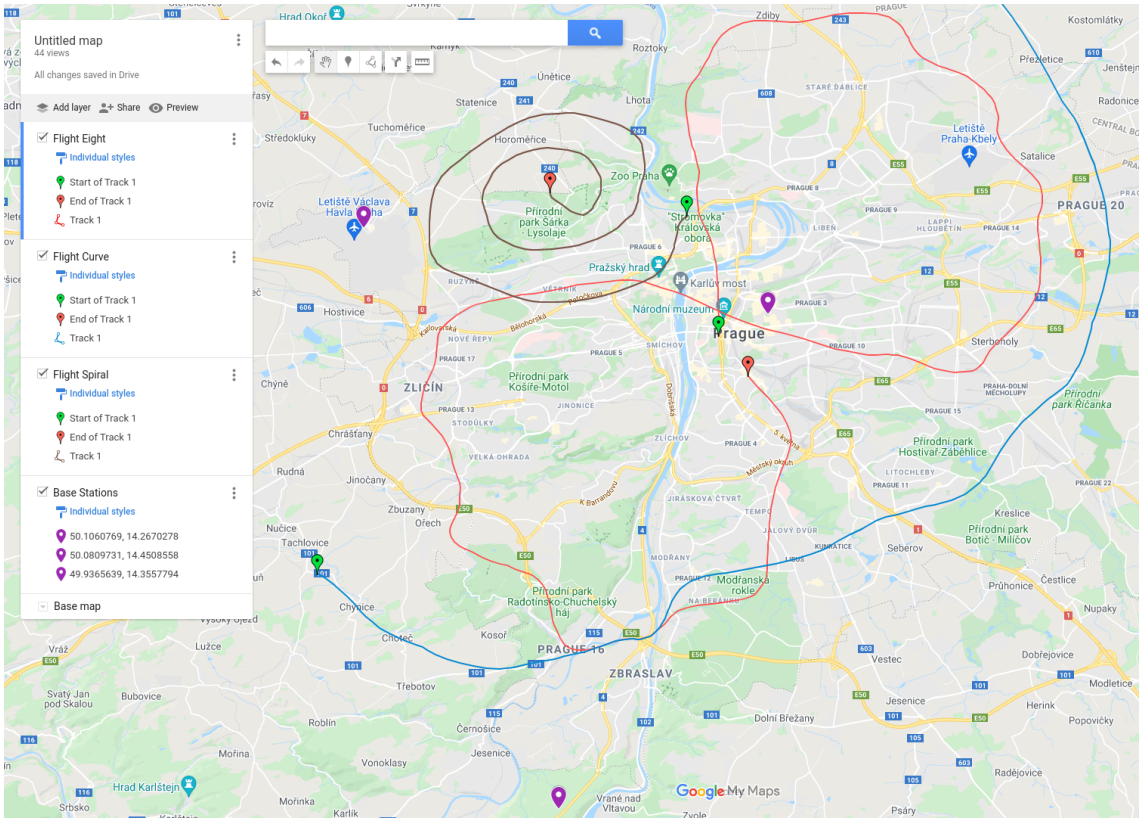

Fig. A.5: Complete view of the scenario, all tracks and base stations, `flipe-beautify` applied, Google Maps [11]

Listing A.3: Output of `flipe-mux` for station ALPHA, first ten messages with their times

```
0000000000019371522 8d49d3d6582184d39cf748cc7e44  1
0000000000421762849 8d49d449580bc4d97cf682347b17  2
0000000000437334419 8d49d1c55809f4c7f4edb1b57bc9  3
0000000000519371470 8d49d3d6582181620d0bcd0788ce  4
0000000000921762781 8d49d449580bc168050b02456efc  5
0000000000937334396 8d49d1c55809f1563101f47a5bfa  6
0000000001019371363 8d49d3d6582184d3ecf749214a26  7
0000000001421762696 8d49d449580bc4d96af67ac01225  8
0000000001437334396 8d49d1c55809f4c7f4edbdb53393  9
0000000001519371287 8d49d3d6582181625d0bceafdc9e  10
```

Listing A.4: Output of `flipe-mux` for station BRAVO, first ten messages with their times

```
0000000000019345735 8d49d3d6582184d39cf748cc7e44  1
0000000000421741652 8d49d449580bc4d97cf682347b17  2
0000000000437352815 8d49d1c55809f4c7f4edb1b57bc9  3
0000000000519345680 8d49d3d6582181620d0bcd0788ce  4
0000000000921741628 8d49d449580bc168050b02456efc  5
0000000000937352726 8d49d1c55809f1563101f47a5bfa  6
0000000001019345650 8d49d3d6582184d3ecf749214a26  7
0000000001421741627 8d49d449580bc4d96af67ac01225  8
0000000001437352638 8d49d1c55809f4c7f4edbdb53393  9
0000000001519345617 8d49d3d6582181625d0bceafdc9e  10
```

Listing A.5: Output of `flipe-mux` for station CHARLIE, first ten messages with their times

```
0000000000019384292 8d49d3d6582184d39cf748cc7e44  1
0000000000421793372 8d49d449580bc4d97cf682347b17  2
0000000000437332958 8d49d1c55809f4c7f4edb1b57bc9  3
0000000000519384614 8d49d3d6582181620d0bcd0788ce  4
0000000000921793274 8d49d449580bc168050b02456efc  5
0000000000937332896 8d49d1c55809f1563101f47a5bfa  6
0000000001019384917 8d49d3d6582184d3ecf749214a26  7
0000000001421793188 8d49d449580bc4d96af67ac01225  8
0000000001437332808 8d49d1c55809f4c7f4edbdb53393  9
0000000001519385231 8d49d3d6582181625d0bceafdc9e  10
```